
synthcity

vanderschaar-lab

Mar 11, 2024

CONTENTS:

1	Features:	1
2	Installation	3
3	Sample Usage	5
3.1	Generic data	5
3.2	Static Survival analysis	6
3.3	Time series	6
3.4	Images	7
4	Methods	9
4.1	Bayesian methods	9
4.2	Generative adversarial networks(GANs)	9
4.3	Variational autoencoders(VAE)	9
4.4	Normalizing Flows	10
4.5	Static Survival analysis methods	10
4.6	Time-Series and Time-Series Survival Analysis methods	10
4.7	Privacy & Fairness	11
4.8	Domain adaptation	12
4.9	Images	12
4.10	Debug methods	12
5	Evaluation metrics	13
6	Tests	17
7	Citing	19
8	Examples	21
8.1	Tutorials	21
9	Dataloaders and Datasets	29
9.1	Datasets and DataLoaders	29
10	Generators	43
10.1	Generators	43
11	Metrics	165
11.1	Metrics and benchmarks	165
12	Advanced Topics	193
12.1	Advanced topics	193

Python Module Index	531
Index	533

FEATURES:

- Easy to extend pluginable architecture.
- Several evaluation metrics for correctness and privacy.
- Several reference models, by type:
 - General purpose: GAN-based (AdsGAN, CTGAN, PATEGAN, DP-GAN), VAE-based (TVAE, RTVAE), Normalizing flows, Bayesian Networks (PrivBayes, BN).
 - Time Series & Time-Series Survival generators: TimeGAN, FourierFlows, TimeVAE.
 - Static Survival Analysis: SurvivalGAN, SurVAE.
 - Privacy-focused: DECAF, DP-GAN, AdsGAN, PATEGAN, PrivBayes.
 - Domain adaptation: RadialGAN.
 - Images: Image ConditionalGAN, Image AdsGAN.
- [Read the docs !](#)
- [Checkout the tutorials!](#)

INSTALLATION

The library can be installed from PyPI using

```
$ pip install synthcity
```

or from source, using

```
$ pip install .
```


SAMPLE USAGE

3.1 Generic data

- List the available general-purpose generators

```
from synthcity.plugins import Plugins

Plugins(categories=["generic", "privacy"]).list()
```

- Load and train a tabular generator

```
from sklearn.datasets import load_diabetes
from synthcity.plugins import Plugins

X, y = load_diabetes(return_X_y=True, as_frame=True)
X["target"] = y

syn_model = Plugins().get("adsgan")

syn_model.fit(X)
```

- Generate new synthetic tabular data

```
syn_model.generate(count = 10)
```

- Benchmark the quality of the plugins

```
# third party
from sklearn.datasets import load_diabetes

# synthcity absolute
from synthcity.benchmark import Benchmarks
from synthcity.plugins.core.constraints import Constraints
from synthcity.plugins.core.data_loader import GenericDataLoader

X, y = load_diabetes(return_X_y=True, as_frame=True)
X["target"] = y

loader = GenericDataLoader(X, target_column="target", sensitive_columns=["sex"])

score = Benchmarks.evaluate(
```

(continues on next page)

(continued from previous page)

```
[
    (f"example_{model}", model, {}) # testname, plugin name, plugin args
    for model in ["adsgan", "ctgan", "tvae"]
],
loader,
synthetic_size=1000,
metrics={"performance": ["linear_model"]},
repeats=3,
)
Benchmarks.print(score)
```

3.2 Static Survival analysis

- List the available generators dedicated to survival analysis

```
from synthcity.plugins import Plugins

Plugins(categories=["generic", "privacy", "survival_analysis"]).list()
```

- Generate new data

```
from lifelines.datasets import load_rossi
from synthcity.plugins.core.data_loader import SurvivalAnalysisDataLoader
from synthcity.plugins import Plugins

X = load_rossi()
data = SurvivalAnalysisDataLoader(
    X,
    target_column="arrest",
    time_to_event_column="week",
)

syn_model = Plugins().get("survival_gan")

syn_model.fit(data)

syn_model.generate(count=10)
```

3.3 Time series

- List the available generators

```
from synthcity.plugins import Plugins

Plugins(categories=["generic", "privacy", "time_series"]).list()
```

- Generate new data

```
# synthcity absolute
from synthcity.plugins import Plugins
from synthcity.plugins.core.data_loader import TimeSeriesDataLoader
from synthcity.utils.datasets.time_series.google_stocks import GoogleStocksDataLoader

static_data, temporal_data, horizons, outcome = GoogleStocksDataLoader().load()
data = TimeSeriesDataLoader(
    temporal_data=temporal_data,
    observation_times=horizons,
    static_data=static_data,
    outcome=outcome,
)

syn_model = Plugins().get("timegan")

syn_model.fit(data)

syn_model.generate(count=10)
```

3.4 Images

Note : The architectures used for generators are not state-of-the-art. For other architectures, consider extending the `suggest_image_generator_discriminator_arch` method from the `convnet.py` module.

- List the available generators

```
from synthcity.plugins import Plugins

Plugins(categories=["images"]).list()
```

- Generate new data ````python` from `synthcity.plugins import Plugins` from `synthcity.plugins.core.data_loader import ImageDataLoader` from `torchvision` import `datasets`

```
dataset = datasets.MNIST(".", download=True) loader = ImageDataLoader(dataset).sample(100)
syn_model = Plugins().get("image_cgan")
syn_model.fit(loader)
syn_img, syn_labels = syn_model.generate(count=10).unpack().numpy()
print(syn_img.shape)
```

```
### Serialization

* Using save/load methods

```python
from synthcity.utils.serialization import save, load
from synthcity.plugins import Plugins

syn_model = Plugins().get("adsgan")

buff = save(syn_model)
```

(continues on next page)

(continued from previous page)

```
reloaded = load(buff)

assert syn_model.name() == reloaded.name()
```

- Saving and loading models from disk

```
from synthcity.utils.serialization import save_to_file, load_from_file
from synthcity.plugins import Plugins

syn_model = Plugins().get("adsgan")

save_to_file('./adsgan_10_epochs.pkl', syn_model)
reloaded = load_from_file('./adsgan_10_epochs.pkl')

assert syn_model.name() == reloaded.name()
```

- Using the Serializable interface

```
from synthcity.plugins import Plugins

syn_model = Plugins().get("adsgan")

buff = syn_model.save()
reloaded = Plugins().load(buff)

assert syn_model.name() == reloaded.name()
```

## METHODS

## 4.1 Bayesian methods

Method	Description	Reference
<b>bayesian-network</b>	A method represents a set of random variables and their conditional dependencies via a directed acyclic graph (DAG), and uses it to sample new data points	<a href="#">pgmpy</a>
<b>privbayes</b>	A differentially private method for releasing high-dimensional data.	<a href="#">PrivBayes: Private Data Release via Bayesian Networks</a>

## 4.2 Generative adversarial networks(GANs)

Method	Description	Reference
<b>ads-gan</b>	A conditional GAN framework that generates synthetic data while minimize patient identifiability that is defined based on the probability of re-identification given the combination of all data on any individual patient	<a href="#">Anonymization Through Data Synthesis Using Generative Adversarial Networks (ADS-GAN)</a>
<b>pate-gan</b>	The method uses the Private Aggregation of Teacher Ensembles (PATE) framework and applies it to GANs, allowing to tightly bound the influence of any individual sample on the model, resulting in tight differential privacy guarantees and thus an improved performance over models with the same guarantees.	<a href="#">PATE-GAN: Generating Synthetic Data with Differential Privacy Guarantees</a>
<b>ct-gan</b>	A conditional generative adversarial network which can handle tabular data.	<a href="#">Modeling Tabular data using Conditional GAN</a>

## 4.3 Variational autoencoders(VAE)

Method	Description	Reference
<b>tvae</b>	A conditional VAE network which can handle tabular data.	<a href="#">Modeling Tabular data using Conditional GAN</a>
<b>rt-vae</b>	A robust variational autoencoder with divergence for tabular data (RTVAE) with mixed categorical and continuous features.	<a href="#">Robust Variational Autoencoder for Tabular Data with Divergence</a>

## 4.4 Normalizing Flows

Method	Description	Reference
<b>nflow</b>	Normalizing Flows are generative models which produce tractable distributions where both sampling and density evaluation can be efficient and exact.	<a href="#">Neural Spline Flows</a>

## 4.5 Static Survival analysis methods

Method	Description	Reference
<b>survival_gan</b>	SurvivalGAN is a generative model that can handle survival data by addressing the imbalance in the censoring and time horizons, using a dedicated mechanism for approximating time to event/censoring from the input and survival function.	—
<b>survival_ctgan</b>	SurvivalGAN version using CTGAN	—
<b>survae</b>	SurvivalGAN version using VAE	—
<b>survival_nflow</b>	SurvivalGAN version using normalizing flows	—

## 4.6 Time-Series and Time-Series Survival Analysis methods

Method	Description	Reference
<b>timegan</b>	TimeGAN is a framework for generating realistic time-series data that combines the flexibility of the unsupervised paradigm with the control afforded by supervised training. Through a learned embedding space jointly optimized with both supervised and adversarial objectives, the network adheres to the dynamics of the training data during sampling.	<a href="#">Time-series Generative Adversarial Networks</a>
<b>fflows</b>	FFlows is an explicit likelihood model based on a novel class of normalizing flows that view time-series data in the frequency-domain rather than the time-domain. The method uses a discrete Fourier transform (DFT) to convert variable-length time-series with arbitrary sampling periods into fixed-length spectral representations, then applies a (data-dependent) spectral filter to the frequency-transformed time-series.	<a href="#">Generative Time-series Modeling with Fourier Flows</a>
<b>probabilistic_ar</b>	Probabilistic AutoRegressive model allows learning multi-type, multivariate timeseries data and later on generate new synthetic data that has the same format and properties as the learned one.	<a href="#">PAR model</a>

## 4.7 Privacy & Fairness

Method	Description	Reference
<b>de- caf</b>	Machine learning models have been criticized for reflecting unfair biases in the training data. Instead of solving this by introducing fair learning algorithms directly, DECAF focuses on generating fair synthetic data, such that any downstream learner is fair. Generating fair synthetic data from unfair data - while remaining truthful to the underlying data-generating process (DGP) - is non-trivial. DECAF is a GAN-based fair synthetic data generator for tabular data. With DECAF, we embed the DGP explicitly as a structural causal model in the input layers of the generator, allowing each variable to be reconstructed conditioned on its causal parents. This procedure enables inference time debiasing, where biased edges can be strategically removed to satisfy user-defined fairness requirements.	DECAF: Generating Fair Synthetic Data Using Causally-Aware Generative Networks
<b>priv- bayes</b>	Differentially private method for releasing high-dimensional data.	PrivBayes: Private Data Release via Bayesian Networks
<b>dp- gan</b>	Differentially Private GAN	Differentially Private Generative Adversarial Network
<b>ads- gan</b>	A conditional GAN framework that generates synthetic data while minimize patient identifiability that is defined based on the probability of re-identification given the combination of all data on any individual patient	Anonymization Through Data Synthesis Using Generative Adversarial Networks (ADS-GAN)
<b>pate- gan</b>	The method uses the Private Aggregation of Teacher Ensembles (PATE) framework and applies it to GANs, allowing to tightly bound the influence of any individual sample on the model, resulting in tight differential privacy guarantees and thus an improved performance over models with the same guarantees.	PATE-GAN: Generating Synthetic Data with Differential Privacy Guarantees

## 4.8 Domain adaptation

Method	Description	Reference
<b>radial-gan</b>	Training complex machine learning models for prediction often requires a large amount of data that is not always readily available. Leveraging these external datasets from related but different sources is, therefore, an essential task if good predictive models are to be built for deployment in settings where data can be rare. RadialGAN is an approach to the problem in which multiple GAN architectures are used to learn to translate from one dataset to another, thereby allowing to augment the target dataset effectively and learning better predictive models than just the target dataset.	<a href="#">RadialGAN: Leveraging multiple datasets to improve target-specific predictive models using Generative Adversarial Networks</a>

## 4.9 Images

Method	Description	Reference
<b>image_cgan</b>	Conditional GAN for generating images	—
<b>image_adsgan</b>	The Adsgan method adapted for image generation	—

## 4.10 Debug methods

Method	Description	Reference
<b>marginal_distributions</b>	A differentially private method that samples from the marginal distributions of the training set	—
<b>uniform_sampler</b>	A differentially private method that uniformly samples from the [min, max] ranges of each column.	—
<b>dummy_sampler</b>	Resample data points from the training set	—



## EVALUATION METRICS

The following table contains the available evaluation metrics:

- **Sanity checks**

Metric	Description	Values
<b>data_mismatch</b>	Average number of columns with datatype(object, real, int) mismatch between the real and synthetic data	0: no datatype mismatch. 1: complete data type mismatch between the datasets.
<b>common_rows_proportion</b>	The proportion of rows in the real dataset present in the synthetic dataset.	0: there are no common rows between the real and synthetic datasets. 1: all the rows in the real dataset are leaked in the synthetic dataset.
<b>nearest_syn_neighbor_distance</b>	Average distance from the real data to the nearest neighbor in the synthetic data	0: all the real rows are leaked in the synthetic dataset. 1: all the synthetic rows are far away from the real dataset.
<b>close_values_probability</b>	The probability of close values between the real and synthetic data.	0: there is no chance to have synthetic rows similar to the real. 1 means that all the synthetic rows are similar to some real rows.
<b>distant_values_probability</b>	Average distance from the real data to the nearest neighbor in the synthetic data	0: no chance to have rows in the synthetic far away from the real data. 1: all the synthetic datapoints are far away from the real data.

- **Statistical tests**

Metric	Description	Values
<b>in-verse_kl_divergence</b>	The average inverse of the Kullback–Leibler Divergence	0: the datasets are from different distributions. 1: the datasets are from the same distribution.
<b>ks_test</b>	The Kolmogorov-Smirnov test	0: the distributions are totally different. 1: the distributions are identical.
<b>chi_squared_test</b>	Chi-squared test p-value. A small value indicates that we can reject the null hypothesis and that the distributions are different.	0: the distributions are different 1: the distributions are identical.
<b>max_mean_discrepancy</b>	Maximum mean discrepancy.	0: The distributions are the same. 1: The distributions are totally different.
<b>jensen-shannon_dist</b>	The Jensen-Shannon distance (metric) between two probability arrays. This is the square root of the Jensen-Shannon divergence.	0: The distributions are the same. 1: The distributions are totally different.
<b>wasserstein_dist</b>	Wasserstein Distance is a measure of the distance between two probability distributions.	0: The distributions are the same.
<b>prdc</b>	Computes precision, recall, density, and coverage given two manifolds.	—
<b>alpha_precision</b>	Evaluate the alpha-precision, beta-recall, and authenticity scores.	—
<b>survival_km_distance</b>	The distance between two Kaplan-Meier plots (survival analysis).	—
<b>fid</b>	The Frechet Inception Distance (FID) calculates the distance between two distributions of images.	—

• Synthetic Data quality

Metric	Description	Values
<b>performance.xgb</b>	Train an XGBoost classifier/regressor/survival model on real data(gt) and synthetic data(syn), and evaluate the performance on the test set.	1 for ideal performance, 0 for worst performance
<b>performance.linear</b>	Train a Linear classifier/regressor/survival model on real data(gt) and the synthetic data and evaluate the performance on test data.	1 for ideal performance, 0 for worst performance
<b>performance.mlp</b>	Train a Neural Net classifier/regressor/survival model on the real data and the synthetic data and evaluate the performance on test data.	1 for ideal performance, 0 for worst performance
<b>performance.feature_rank_distance</b>	Train a model on the synthetic data and a model on the real data. Compute the rank distance between the importance(kendalltau or spearman)	1: similar ranks in the feature importance. 0: uncorrelated feature importance
<b>detection_gmm</b>	Train a GaussianMixture model to differentiate the synthetic data from the real data.	0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.
<b>detection_xgb</b>	Train an XGBoost model to differentiate the synthetic data from the real data.	0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.
<b>detection_mlp</b>	Train a Neural net to differentiate the synthetic data from the real data.	0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.
<b>detection_linear</b>	Train a Linear model to differentiate the synthetic data from the real data.	0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.

- Privacy metrics

*Quasi-identifiers* : pieces of information that are not of themselves unique identifiers, but are sufficiently well correlated with an entity that they can be combined with other quasi-identifiers to create a unique identifier.

Metric	Description	Values
<b>k_anonymization</b>	The minimum value k which satisfies the k-anonymity rule: each record is similar to at least another k-1 other records on the potentially identifying variables.	Reported on both the real and synthetic data.
<b>l_diversity</b>	The minimum value l which satisfies the l-diversity rule: every generalized block has to contain at least l different sensitive values.	Reported on both the real and synthetic data.
<b>kmap</b>	The minimum value k which satisfies the k-map rule: every combination of values for the quasi-identifiers appears at least k times in the reidentification(synthetic) dataset.	Reported on both the real and synthetic data.
<b>delta_presence</b>	The maximum re-identification risk for the real dataset from the synthetic dataset.	0 for no risk.
<b>identifiability_score</b>	The re-identification score on the real dataset from the synthetic dataset.	— ]
<b>sensitive_data_reidentification_xgb</b>	Sensitive data prediction from the quasi-identifiers using an XGBoost.	0 for no risk.
<b>sensitive_data_reidentification_mlp</b>	Sensitive data prediction from the quasi-identifiers using a Neural Net.	0 for no risk.



## TESTS

Install the testing dependencies using

```
pip install .[testing]
```

The tests can be executed using

```
pytest -vsx
```



## CITING

If you use this code, please cite the associated paper:

```
@misc{https://doi.org/10.48550/arxiv.2301.07573,
 doi = {10.48550/ARXIV.2301.07573},
 url = {https://arxiv.org/abs/2301.07573},
 author = {Qian, Zhaozhi and Cebere, Bogdan-Constantin and van der Schaar, Mihaela},
 keywords = {Machine Learning (cs.LG), Artificial Intelligence (cs.AI), FOS: Computer
↪ and information sciences, FOS: Computer and information sciences},
 title = {Synthcity: facilitating innovative use cases of synthetic data in different
↪ data modalities},
 year = {2023},
 copyright = {Creative Commons Attribution 4.0 International}
}
```





## EXAMPLES

## 8.1 Tutorials

### 8.1.1 Getting started

### 8.1.2 General-purpose generators

#### `synthcity.plugins.generic.plugin_goggle` module

Reference: “GOGGLE: Generative Modelling for Tabular Data by Learning Relational Structure” Authors: Tennison Liu, Zhaozhi Qian, Jeroen Berrevoets, Mihaela van der Schaar

```
class GOGGLEPlugin(n_iter: int = 1000, encoder_dim: int = 64, encoder_l: int = 2, het_encoding: bool = True,
 encoder_nonlin: str = 'tanh', decoder_nonlin: str = 'tanh', decoder_dim: int = 64,
 decoder_l: int = 2, data_encoder_max_clusters: int = 10, threshold: float = 0.1,
 decoder_arch: str = 'gcn', graph_prior: Optional[numpy.ndarray] = None, prior_mask:
 Optional[numpy.ndarray] = None, alpha: float = 0.1, beta: float = 0.1, iter_opt: bool =
 True, learning_rate: float = 0.005, weight_decay: float = 0.001, batch_size: int = 32,
 patience: int = 50, logging_epoch: int = 100, device: Union[str, torch.device] =
 device(type='cpu'), random_state: int = 0, sampling_patience: int = 500, workspace:
 pathlib.Path = PosixPath('workspace'), compress_dataset: bool = False,
 dataloader_sampler: Optional[torch.utils.data.sampler.Sampler] = None, **kwargs: Any)
```

Bases: `synthcity.plugins.core.plugin.Plugin`

Args:

#### Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("goggle", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

```
class Config
```

```
Bases: object
```

```
arbitrary_types_allowed = True
```

```
validate_assignment = True
```

```
fit(X: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame], *args: Any,
 **kwargs: Any) → Any
```

Training method the synthetic data plugin.

#### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "==", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()
```

Returns self

```
classmethod fqdn() → str
```

The Fully-Qualified name of the plugin.

**generate**(*count*: *Optional[int] = None*, *constraints*: *Optional[synthcity.plugins.core.constraints.Constraints]* = *None*, *random\_state*: *Optional[int] = None*, *\*\*kwargs*: *Any*) → *synthcity.plugins.core.data\_loader.DataLoader*

Synthetic data generation method.

#### Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

#### Valid Operations:

- "<", "lt": less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt": greater than <value>
- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

#### Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "=", 0),
>>>]
>>>)
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(*\*\*kwargs*: *Any*) → List[*synthcity.plugins.core.distribution.Distribution*]

Returns the hyperparameter space for the derived plugin.

**static load**(*buff*: *bytes*) → *Any*

**static load\_dict**(*representation: dict*) → Any

**static name**() → str  
The name of the plugin.

**plot**(*plt: Any, X: synthcity.plugins.core.dataloader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any*) → Any  
Plot the real-synthetic distributions.

**Parameters**

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(*trial: Any, \*args: Any, \*\*kwargs: Any*) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(*path: pathlib.Path*) → bytes

**schema**() → *synthcity.plugins.core.schema.Schema*  
The reference schema

**schema\_includes**(*other: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame]*) → bool  
Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema**() → *synthcity.plugins.core.schema.Schema*  
The internal schema

**static type**() → str  
The type of the plugin.

**static version**() → str  
API version

**plugin**

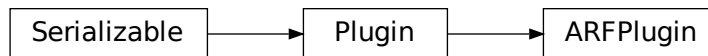
alias of *synthcity.plugins.generic.plugin\_goggle.GOGGLEPlugin*

## **synthcity.plugins.generic.plugin\_arf module**

Reference: “Adversarial random forests for density estimation and generative modeling” Authors: David S. Watson, Kristin Blesch, Jan Kapar, and Marvin N. Wright

**class ARFPlugin**(*num\_trees: int = 30, delta: int = 0, max\_iters: int = 10, early\_stop: bool = True, verbose: bool = True, min\_node\_size: int = 5, device: Union[str, torch.device] = device(type='cpu'), random\_state: int = 0, sampling\_patience: int = 500, workspace: pathlib.Path = PosixPath('workspace'), compress\_dataset: bool = False, \*\*kwargs: Any*)

Bases: *synthcity.plugins.core.plugin.Plugin*



Args:

### Example

```

>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("arf")
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)

```

### class Config

Bases: object

**arbitrary\_types\_allowed** = True

**validate\_assignment** = True

**fit**(X: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any

Training method the synthetic data plugin.

### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```

>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import _
↳ GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>

```

(continues on next page)

(continued from previous page)

```

>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "=", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()

```

**Returns** self**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any) → synthcity.plugins.core.data\_loader.DataLoader

Synthetic data generation method.

**Parameters**

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

**Valid Operations:**

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>

- "==" , "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "==", 0),
>>>]
>>>)
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**static name**() → str  
The name of the plugin.

**plot**(plt: Any, X: synthcity.plugins.core.dataloader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any) → Any  
Plot the real-synthetic distributions.

**Parameters**

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(trial: Any, \*args: Any, \*\*kwargs: Any) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(path: pathlib.Path) → bytes

**schema**() → synthcity.plugins.core.schema.Schema  
The reference schema

**schema\_includes**(*other*: Union[synthcity.plugins.core.dataloader.DataLoader,  
pandas.core.frame.DataFrame]) → bool

Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema**() → synthcity.plugins.core.schema.Schema

The internal schema

**static type**() → str

The type of the plugin.

**static version**() → str

API version

**plugin**

alias of *synthcity.plugins.generic.plugin\_arf.ARFPlugin*

**synthcity.plugins.generic.plugin\_great** module

### 8.1.3 Time-series generators

### 8.1.4 Privacy-related generators

### 8.1.5 Domain adaptation generators

### 8.1.6 Images



## DATALOADERS AND DATASETS

### 9.1 Datasets and DataLoaders

#### 9.1.1 Dataloaders and Datasets

**synthcity.plugins.core.dataloader module**

```
class DataLoader(data_type: str, data: Any, static_features: List[str] = [], temporal_features: List[str] = [],
 sensitive_features: List[str] = [], important_features: List[str] = [], outcome_features:
 List[str] = [], train_size: float = 0.8, random_state: int = 0, **kwargs: Any)
```

Bases: object

DataLoader

Base class for all data loaders.

**Each derived class must implement the following methods:** `unpack()` - a method that unpacks the columns and returns features and labels (X, y). `decorate()` - a method that creates a new instance of `DataLoader` by decorating the input data with the same `DataLoader` properties (e.g. sensitive features, target column, etc.) `dataframe()` - a method that returns the pandas dataframe that contains all features and samples `numpy()` - a method that returns the numpy array that contains all features and samples `info()` - a method that returns a dictionary of `DataLoader` information `__len__()` - a method that returns the number of samples in the `DataLoader` `satisfies()` - a method that tests if the current `DataLoader` satisfies the constraint provided `match()` - a method that returns a new `DataLoader` where the provided constraints are met `from_info()` - a static method that creates a `DataLoader` from the data and the information dictionary `sample()` - returns a new `DataLoader` that contains a random subset of N samples `drop()` - returns a new `DataLoader` with a list of columns dropped `__getitem__()` - getting features by names `__setitem__()` - setting features by names `train()` - returns a `DataLoader` containing the training set `test()` - returns a `DataLoader` containing the testing set `fillna()` - returns a `DataLoader` with NaN filled by the provided number(s)

If any method implementation is missing, the class constructor will fail.

**Constructor Args:**

**data\_type: str** The type of `DataLoader`, currently supports “generic”, “time\_series” and “survival”.

**data: Any** The object that contains the data

**static\_features: List[str]** List of feature names that are static features (as opposed to temporal features).

**temporal\_features:** List of feature names that are temporal features, i.e. observed over time.

**sensitive\_features: List[str]** Name of sensitive features.

**important\_features: List[str]** Default: None. Only relevant for SurvivalGAN method.

**outcome\_features:** The feature name that provides labels for downstream tasks.

**abstract property columns: list**

**compress()** → Tuple[synthcity.plugins.core.dataloader.DataLoader, Dict]

**abstract compression\_protected\_features()** → list

**abstract dataframe()** → pandas.core.frame.DataFrame

**decode(encoders: Dict[str, Any])** → synthcity.plugins.core.dataloader.DataLoader

**decompress(context: Dict)** → synthcity.plugins.core.dataloader.DataLoader

**abstract decorate(data: Any)** → synthcity.plugins.core.dataloader.DataLoader

**domain()** → Optional[str]

**abstract drop(columns: list = [])** → synthcity.plugins.core.dataloader.DataLoader

**encode(encoders: Optional[Dict[str, Any]] = None)** → Tuple[synthcity.plugins.core.dataloader.DataLoader, Dict]

**abstract fillna(value: Any)** → synthcity.plugins.core.dataloader.DataLoader

**abstract static from\_info(data: pandas.core.frame.DataFrame, info: dict)** → synthcity.plugins.core.dataloader.DataLoader

**abstract get\_fairness\_column()** → Union[str, Any]

**hash()** → str

**abstract info()** → dict

**abstract is\_tabular()** → bool

**abstract match(constraints: synthcity.plugins.core.constraints.Constraints)** → synthcity.plugins.core.dataloader.DataLoader

**abstract numpy()** → numpy.ndarray

**raw()** → Any

**abstract sample(count: int, random\_state: int = 0)** → synthcity.plugins.core.dataloader.DataLoader

**abstract satisfies(constraints: synthcity.plugins.core.constraints.Constraints)** → bool

**abstract property shape: tuple**

**abstract test()** → synthcity.plugins.core.dataloader.DataLoader

**abstract train()** → synthcity.plugins.core.dataloader.DataLoader

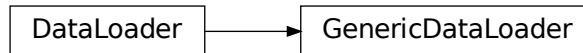
**type()** → str

**abstract unpack(as\_numpy: bool = False, pad: bool = False)** → Any

**property values: numpy.ndarray**

```
class GenericDataLoader(data: Union[pandas.core.frame.DataFrame, list, numpy.ndarray], sensitive_features:
 List[str] = [], important_features: List[str] = [], target_column: Optional[str] =
 None, fairness_column: Optional[str] = None, domain_column: Optional[str] =
 None, random_state: int = 0, train_size: float = 0.8, **kwargs: Any)
```

Bases: `synthcity.plugins.core.data_loader.DataLoader`



Data loader for generic tabular data.

#### Constructor Args:

**data:** `Union[pd.DataFrame, list, np.ndarray]` The dataset. Either a Pandas DataFrame or a Numpy Array.

**sensitive\_features:** `List[str]` Name of sensitive features.

**important\_features:** `List[str]` Default: None. Only relevant for SurvivalGAN method.

**target\_column:** `Optional[str]` The feature name that provides labels for downstream tasks.

**fairness\_column:** `Optional[str]` Optional fairness column label, used for fairness benchmarking.

**domain\_column:** `Optional[str]` Optional domain label, used for domain adaptation algorithms.

**random\_state:** `int` Defaults to zero.

#### Example

```
>>> from sklearn.datasets import load_diabetes
>>> from synthcity.plugins.core.data_loader import GenericDataLoader
>>> X, y = load_diabetes(return_X_y=True, as_frame=True)
>>> X["target"] = y
>>> # Important note: preprocessing data with OneHotEncoder or StandardScaler is
↳not needed or recommended.
>>> # Synthcity handles feature encoding and standardization internally.
>>> loader = GenericDataLoader(X, target_column="target", sensitive_columns=["sex"],
↳)
```

**property columns:** `list`

**compress()** → `Tuple[synthcity.plugins.core.data_loader.DataLoader, Dict]`

**compression\_protected\_features()** → `list`

**dataframe()** → `pandas.core.frame.DataFrame`

**decode(encoders: Dict[str, Any])** → `synthcity.plugins.core.data_loader.DataLoader`

**decompress(context: Dict)** → `synthcity.plugins.core.data_loader.DataLoader`

**decorate(data: Any)** → `synthcity.plugins.core.data_loader.DataLoader`

**domain()** → `Optional[str]`

```
drop(columns: list = []) → synthcity.plugins.core.dataloader.DataLoader
encode(encoders: Optional[Dict[str, Any]] = None) → Tuple[synthcity.plugins.core.dataloader.DataLoader,
Dict]
fillna(value: Any) → synthcity.plugins.core.dataloader.DataLoader
static from_info(data: pandas.core.frame.DataFrame, info: dict) →
 synthcity.plugins.core.dataloader.GenericDataLoader
get_fairness_column() → Union[str, Any]
hash() → str
info() → dict
is_tabular() → bool
match(constraints: synthcity.plugins.core.constraints.Constraints) →
 synthcity.plugins.core.dataloader.DataLoader
numpy() → numpy.ndarray
raw() → Any
sample(count: int, random_state: int = 0) → synthcity.plugins.core.dataloader.DataLoader
satisfies(constraints: synthcity.plugins.core.constraints.Constraints) → bool
property shape: tuple
test() → synthcity.plugins.core.dataloader.DataLoader
train() → synthcity.plugins.core.dataloader.DataLoader
type() → str
unpack(as_numpy: bool = False, pad: bool = False) → Any
property values: numpy.ndarray
class ImageDataLoader(data: Union[torch.utils.data.dataset.Dataset, Tuple[torch.Tensor, torch.Tensor]], height:
 int = 32, width: Optional[int] = None, random_state: int = 0, train_size: float = 0.8,
 **kwargs: Any)
 Bases: synthcity.plugins.core.dataloader.DataLoader
```



Data loader for generic image data.

**Constructor Args:**

**data: torch.utils.data.Dataset or torch.Tensor** The image dataset or a tuple of (tensor images, tensor labels)

**random\_state: int** Defaults to zero.

**height: int. Default = 32** Height to use internally

**width: Optional[int]** Optional width to use internally. If None, it is used the same value as height.

**train\_size: float = 0.8** Train dataset ratio.

### Example

```
>>> dataset = datasets.MNIST(".", download=True)
>>>
>>> loader = ImageDataLoader(
>>> data=dataset,
>>> train_size=0.8,
>>> height=32,
>>> width=32,
>>>)
```

**property columns: list**

**compress()** → Tuple[synthcity.plugins.core.data\_loader.DataLoader, Dict]

**compression\_protected\_features()** → list

**dataframe()** → pandas.core.frame.DataFrame

**decode(encoders: Dict[str, Any])** → synthcity.plugins.core.data\_loader.DataLoader

**decompress(context: Dict)** → synthcity.plugins.core.data\_loader.DataLoader

**decorate(data: Any)** → synthcity.plugins.core.data\_loader.DataLoader

**domain()** → Optional[str]

**drop(columns: list = [])** → synthcity.plugins.core.data\_loader.DataLoader

**encode(encoders: Optional[Dict[str, Any]] = None)** → Tuple[synthcity.plugins.core.data\_loader.DataLoader, Dict]

**fillna(value: Any)** → synthcity.plugins.core.data\_loader.DataLoader

**static from\_info(data: torch.utils.data.dataset.Dataset, info: dict)** → synthcity.plugins.core.data\_loader.ImageDataLoader

**get\_fairness\_column()** → None  
Not implemented for ImageDataLoader

**hash()** → str

**info()** → dict

**is\_tabular()** → bool

**match(constraints: synthcity.plugins.core.constraints.Constraints)** → synthcity.plugins.core.data\_loader.DataLoader

**numpy()** → numpy.ndarray

**raw()** → Any

**sample(count: int, random\_state: int = 0)** → synthcity.plugins.core.data\_loader.DataLoader

**satisfies(constraints: synthcity.plugins.core.constraints.Constraints)** → bool

**property shape: tuple**

**test()** → synthcity.plugins.core.data\_loader.DataLoader

**train()** → *synthcity.plugins.core.dataloader.DataLoader*

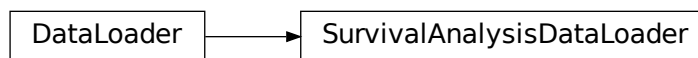
**type()** → str

**unpack**(*as\_numpy: bool = False, pad: bool = False*) → Any

**property values:** `numpy.ndarray`

```
class SurvivalAnalysisDataLoader(data: pandas.core.frame.DataFrame, time_to_event_column: str,
 target_column: str, time_horizons: list = [], sensitive_features: List[str]
 = [], important_features: List[str] = [], fairness_column: Optional[str]
 = None, random_state: int = 0, train_size: float = 0.8, **kwargs: Any)
```

Bases: *synthcity.plugins.core.dataloader.DataLoader*



Data Loader for Survival Analysis Data

#### Constructor Args:

**data:** `Union[pd.DataFrame, list, np.ndarray]` The dataset. Either a Pandas DataFrame or a Numpy Array.

**time\_to\_event\_column:** `str` Survival Analysis specific time-to-event feature

**target\_column:** `str` The outcome: event or censoring.

**sensitive\_features:** `List[str]` Name of sensitive features.

**important\_features:** `List[str]` Default: None. Only relevant for SurvivalGAN method.

**target\_column:** `str` The feature name that provides labels for downstream tasks.

**fairness\_column:** `Optional[str]` Optional fairness column label, used for fairness benchmarking.

**domain\_column:** `Optional[str]` Optional domain label, used for domain adaptation algorithms.

**random\_state:** `int` Defaults to zero.

**train\_size:** `float` The ratio to use for train splits.

#### Example

```
>>> TODO
```

**property columns:** `list`

**compress()** → `Tuple[synthcity.plugins.core.dataloader.DataLoader, Dict]`

**compression\_protected\_features()** → `list`

**dataframe()** → `pandas.core.frame.DataFrame`

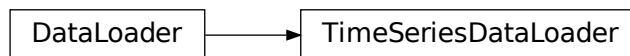
**decode**(*encoders: Dict[str, Any]*) → *synthcity.plugins.core.dataloader.DataLoader*

**decompress**(*context: Dict*) → *synthcity.plugins.core.dataloader.DataLoader*

```

decorate(data: Any) → synthcity.plugins.core.dataloader.DataLoader
domain() → Optional[str]
drop(columns: list = []) → synthcity.plugins.core.dataloader.DataLoader
encode(encoders: Optional[Dict[str, Any]] = None) → Tuple[synthcity.plugins.core.dataloader.DataLoader,
 Dict]
fillna(value: Any) → synthcity.plugins.core.dataloader.DataLoader
static from_info(data: pandas.core.frame.DataFrame, info: dict) →
 synthcity.plugins.core.dataloader.DataLoader
get_fairness_column() → Union[str, Any]
hash() → str
info() → dict
is_tabular() → bool
match(constraints: synthcity.plugins.core.constraints.Constraints) →
 synthcity.plugins.core.dataloader.DataLoader
numpy() → numpy.ndarray
raw() → Any
sample(count: int, random_state: int = 0) → synthcity.plugins.core.dataloader.DataLoader
satisfies(constraints: synthcity.plugins.core.constraints.Constraints) → bool
property shape: tuple
test() → synthcity.plugins.core.dataloader.DataLoader
train() → synthcity.plugins.core.dataloader.DataLoader
type() → str
unpack(as_numpy: bool = False, pad: bool = False) → Any
property values: numpy.ndarray
class TimeSeriesDataLoader(temporal_data: List[pandas.core.frame.DataFrame], observation_times: List,
 outcome: Optional[pandas.core.frame.DataFrame] = None, static_data:
 Optional[pandas.core.frame.DataFrame] = None, sensitive_features: List[str] =
 [], important_features: List[str] = [], fairness_column: Optional[str] = None,
 random_state: int = 0, train_size: float = 0.8, seq_offset: int = 0, **kwargs: Any)
 Bases: synthcity.plugins.core.dataloader.DataLoader

```



Data Loader for Time Series Data

**Constructor Args:**

**temporal data: List[pd.DataFrame]** The temporal data. A list of pandas DataFrames

**observation\_times:** **List** List of arrays mapping directly to index of each dataframe in `temporal_data`

**outcome:** **Optional[pd.DataFrame] = None** pandas DataFrame thatn can be anything (eg, labels, regression outcome)

**static\_data:** **Optional[pd.DataFrame] = None** pandas DataFrame mapping directly to index of each dataframe in `temporal_data`

**sensitive\_features:** **List[str]** Name of sensitive features

**important\_features** **List[str]** Default: None. Only relevant for SurvivalGAN method

**fairness\_column:** **Optional[str]** Optional fairness column label, used for fairness benchmarking.

**random\_state:** **int** Defaults to zero.

### Example

```
>>> TODO
```

```
property columns: list
compress() → Tuple[synthcity.plugins.core.dataloader.DataLoader, Dict]
compression_protected_features() → list
dataframe() → pandas.core.frame.DataFrame
decode(encoders: Dict[str, Any]) → synthcity.plugins.core.dataloader.DataLoader
decompress(context: Dict) → synthcity.plugins.core.dataloader.DataLoader
decorate(data: Any) → synthcity.plugins.core.dataloader.DataLoader
domain() → Optional[str]
drop(columns: list = []) → synthcity.plugins.core.dataloader.DataLoader
encode(encoders: Optional[Dict[str, Any]] = None) → Tuple[synthcity.plugins.core.dataloader.DataLoader, Dict]
static extract_masked_features(full_temporal_features: list) → tuple
fillna(value: Any) → synthcity.plugins.core.dataloader.DataLoader
filter_ids(ids_list: list) → pandas.core.frame.DataFrame
static from_info(data: pandas.core.frame.DataFrame, info: dict) → synthcity.plugins.core.dataloader.DataLoader
get_fairness_column() → Union[str, Any]
hash() → str
ids() → list
info() → dict
is_tabular() → bool
static mask_temporal_data(temporal_data: List[pandas.core.frame.DataFrame], observation_times: List, fill: Any = 0) → Any
match(constraints: synthcity.plugins.core.constraints.Constraints) → synthcity.plugins.core.dataloader.DataLoader
```



**numpy()** → `numpy.ndarray`

**static pack\_raw\_data**(*static\_data*: `Optional[pandas.core.frame.DataFrame]`, *temporal\_data*: `List[pandas.core.frame.DataFrame]`, *observation\_times*: `List`, *outcome*: `Optional[pandas.core.frame.DataFrame]`, *fill*: `Any = nan`, *seq\_offset*: `int = 0`) → `pandas.core.frame.DataFrame`

**static pad\_and\_mask**(*static\_data*: `Optional[pandas.core.frame.DataFrame]`, *temporal\_data*: `List[pandas.core.frame.DataFrame]`, *observation\_times*: `List`, *outcome*: `Optional[pandas.core.frame.DataFrame]`, *only\_features*: `Any = False`, *fill*: `Any = 0`) → `Any`

**static pad\_raw\_data**(*static\_data*: `Optional[pandas.core.frame.DataFrame]`, *temporal\_data*: `List[pandas.core.frame.DataFrame]`, *observation\_times*: `List`, *outcome*: `Optional[pandas.core.frame.DataFrame]`) → `Any`

**static pad\_raw\_features**(*static\_data*: `Optional[pandas.core.frame.DataFrame]`, *temporal\_data*: `List[pandas.core.frame.DataFrame]`, *observation\_times*: `List`, *outcome*: `Optional[pandas.core.frame.DataFrame]`) → `Any`

**raw()** → `Any`

**property raw\_columns**: `list`

**sample**(*count*: `int`, *random\_state*: `int = 0`) → `synthcity.plugins.core.dataloader.DataLoader`

**satisfies**(*constraints*: `synthcity.plugins.core.constraints.Constraints`) → `bool`

**static sequential\_view**(*static\_data*: `Optional[pandas.core.frame.DataFrame]`, *temporal\_data*: `List[pandas.core.frame.DataFrame]`, *observation\_times*: `List`, *outcome*: `Optional[pandas.core.frame.DataFrame]`, *id\_col*: `str = 'seq_id'`, *time\_id\_col*: `str = 'seq_time_id'`, *seq\_offset*: `int = 0`) → `Tuple[pandas.core.frame.DataFrame, dict]`

**property shape**: `tuple`

**test()** → `synthcity.plugins.core.dataloader.DataLoader`

**train()** → `synthcity.plugins.core.dataloader.DataLoader`

**type()** → `str`

**static unique\_temporal\_features**(*temporal\_data*: `List[pandas.core.frame.DataFrame]`) → `List`

**static unmask\_temporal\_data**(*temporal\_data*: `List[pandas.core.frame.DataFrame]`, *observation\_times*: `List`, *fill*: `Any = nan`) → `Any`

**unpack**(*as\_numpy*: `bool = False`, *pad*: `bool = False`) → `Any`

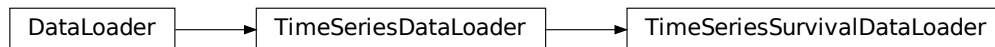
**unpack\_and\_decorate**(*data*: `pandas.core.frame.DataFrame`) → `synthcity.plugins.core.dataloader.DataLoader`

**static unpack\_raw\_data**(*data*: `pandas.core.frame.DataFrame`, *info*: `dict`) → `Tuple[Optional[pandas.core.frame.DataFrame], List[pandas.core.frame.DataFrame], List, Optional[pandas.core.frame.DataFrame]]`

**property values**: `numpy.ndarray`

```
class TimeSeriesSurvivalDataLoader(temporal_data: List[pandas.core.frame.DataFrame],
 observation_times: Union[List, numpy.ndarray,
 pandas.core.series.Series], T: Union[pandas.core.series.Series,
 numpy.ndarray], E: Union[pandas.core.series.Series, numpy.ndarray],
 static_data: Optional[pandas.core.frame.DataFrame] = None,
 sensitive_features: List[str] = [], important_features: List[str] = [],
 time_horizons: list = [], fairness_column: Optional[str] = None,
 random_state: int = 0, train_size: float = 0.8, seq_offset: int = 0,
 **kwargs: Any)
```

Bases: [synthcity.plugins.core.data\\_loader.TimeSeriesDataLoader](#)



Data loader for Time series survival data

#### Constructor Args:

**temporal\_data: List[pd.DataFrame]** The temporal data. A list of pandas DataFrames.

**observation\_times: List** List of arrays mapping directly to index of each dataframe in temporal\_data

**T: Union[pd.Series, np.ndarray, pd.Series]** Time-to-event data

**E: Union[pd.Series, np.ndarray, pd.Series]** E is censored/event data

**static\_data Optional[pd.DataFrame] = None** pandas DataFrame of static features for each subject

**sensitive\_features: List[str]** Name of sensitive features

**important\_features: List[str]** Default: None. Only relevant for SurvivalGAN method.

**fairness\_column: Optional[str]** Optional fairness column label, used for fairness benchmarking.

**random\_state. int** Defaults to zero.

#### Example

```
>>> TODO
```

```
property columns: list
compress() → Tuple[synthcity.plugins.core.data_loader.DataLoader, Dict]
compression_protected_features() → list
dataframe() → pandas.core.frame.DataFrame
decode(encoders: Dict[str, Any]) → synthcity.plugins.core.data_loader.DataLoader
decompress(context: Dict) → synthcity.plugins.core.data_loader.DataLoader
decorate(data: Any) → synthcity.plugins.core.data_loader.DataLoader
domain() → Optional[str]
drop(columns: list = []) → synthcity.plugins.core.data_loader.DataLoader
```

```

encode(encoders: Optional[Dict[str, Any]] = None) → Tuple[synthcity.plugins.core.dataloader.DataLoader,
Dict]

static extract_masked_features(full_temporal_features: list) → tuple

fillna(value: Any) → synthcity.plugins.core.dataloader.DataLoader

filter_ids(ids_list: list) → pandas.core.frame.DataFrame

static from_info(data: pandas.core.frame.DataFrame, info: dict) →
 synthcity.plugins.core.dataloader.DataLoader

get_fairness_column() → Union[str, Any]

hash() → str

ids() → list

info() → dict

is_tabular() → bool

static mask_temporal_data(temporal_data: List[pandas.core.frame.DataFrame], observation_times:
 List, fill: Any = 0) → Any

match(constraints: synthcity.plugins.core.constraints.Constraints) →
 synthcity.plugins.core.dataloader.DataLoader

numpy() → numpy.ndarray

static pack_raw_data(static_data: Optional[pandas.core.frame.DataFrame], temporal_data:
 List[pandas.core.frame.DataFrame], observation_times: List, outcome:
 Optional[pandas.core.frame.DataFrame], fill: Any = nan, seq_offset: int = 0) →
 pandas.core.frame.DataFrame

static pad_and_mask(static_data: Optional[pandas.core.frame.DataFrame], temporal_data:
 List[pandas.core.frame.DataFrame], observation_times: List, outcome:
 Optional[pandas.core.frame.DataFrame], only_features: Any = False, fill: Any = 0)
 → Any

static pad_raw_data(static_data: Optional[pandas.core.frame.DataFrame], temporal_data:
 List[pandas.core.frame.DataFrame], observation_times: List, outcome:
 Optional[pandas.core.frame.DataFrame]) → Any

static pad_raw_features(static_data: Optional[pandas.core.frame.DataFrame], temporal_data:
 List[pandas.core.frame.DataFrame], observation_times: List, outcome:
 Optional[pandas.core.frame.DataFrame]) → Any

raw() → Any

property raw_columns: list

sample(count: int, random_state: int = 0) → synthcity.plugins.core.dataloader.DataLoader

satisfies(constraints: synthcity.plugins.core.constraints.Constraints) → bool

static sequential_view(static_data: Optional[pandas.core.frame.DataFrame], temporal_data:
 List[pandas.core.frame.DataFrame], observation_times: List, outcome:
 Optional[pandas.core.frame.DataFrame], id_col: str = 'seq_id', time_id_col: str
 = 'seq_time_id', seq_offset: int = 0) → Tuple[pandas.core.frame.DataFrame,
dict]

property shape: tuple

test() → synthcity.plugins.core.dataloader.DataLoader

```

```
train() → synthcity.plugins.core.data_loader.DataLoader
type() → str
static unique_temporal_features(temporal_data: List[pandas.core.frame.DataFrame]) → List
static unmask_temporal_data(temporal_data: List[pandas.core.frame.DataFrame], observation_times:
 List, fill: Any = nan) → Any
unpack(as_numpy: bool = False, pad: bool = False) → Any
unpack_and_decorate(data: pandas.core.frame.DataFrame) →
 synthcity.plugins.core.data_loader.DataLoader
static unpack_raw_data(data: pandas.core.frame.DataFrame, info: dict) →
 Tuple[Optional[pandas.core.frame.DataFrame],
 List[pandas.core.frame.DataFrame], List,
 Optional[pandas.core.frame.DataFrame]]

property values: numpy.ndarray
create_from_info(data: Union[pandas.core.frame.DataFrame, torch.utils.data.dataset.Dataset], info: dict) →
 synthcity.plugins.core.data_loader.DataLoader
Helper for creating a DataLoader from existing information.
```

### **synthcity.plugins.core.dataset module**

```
class ConditionalDataset(data: torch.utils.data.dataset.Dataset, cond: Optional[torch.Tensor] = None)
 Bases: torch.utils.data.dataset.Dataset
```

Helper dataset for wrapping existing datasets with custom tensors

#### **Parameters**

- **data** – torch.Dataset
- **cond** – Optional Tensor

```
class FlexibleDataset(data: torch.utils.data.dataset.Dataset, transform:
 Optional[torch.nn.modules.module.Module] = None, indices: Optional[List] = None)
 Bases: torch.utils.data.dataset.Dataset
```

Helper dataset wrapper for post-processing or transforming another dataset. Used for controlling the image sizes for the synthcity models.

The class supports adding custom transforms to existing datasets, and to subsample a set of indices.

#### **Parameters**

- **data** – torch.Dataset
- **transform** – An optional list of transforms
- **indices** – An optional list of indices to subsample

```
filter_indices(indices: List[int]) → synthcity.plugins.core.dataset.FlexibleDataset
labels() → numpy.ndarray
numpy() → Tuple[numpy.ndarray, numpy.ndarray]
shape() → Tuple
tensors() → Tuple[torch.Tensor, torch.Tensor]
```

**class NumpyDataset**(*X: numpy.ndarray, y: numpy.ndarray*)

Bases: torch.utils.data.dataset.Dataset

Helper class for wrapping Numpy arrays in torch Datasets :param X: np.ndarray :param y: np.ndarray

**class TensorDataset**(*images: torch.Tensor, targets: Optional[torch.Tensor]*)

Bases: torch.utils.data.dataset.Dataset

Helper dataset for wrapping existing tensors

**Parameters**

- **images** – Tensor
- **targets** – Tensor

**labels()** → Optional[numpy.ndarray]



## GENERATORS

### 10.1 Generators

#### 10.1.1 General purpose

`synthcity.plugins.generic.plugin_bayesian_network` module

**Reference:** Jankan, Ankur and Panda, Abinash, “pgmpy: Probabilistic graphical models using python,”  
Proceedings of the 14th Python in Science Conference (SCIPY 2015), 2015.

```
class BayesianNetworkPlugin(struct_learning_n_iter: int = 1000, struct_learning_search_method: str =
 'tree_search', struct_learning_score: str = 'k2', struct_max_indegree: int = 4,
 encoder_max_clusters: int = 10, encoder_noise_scale: float = 0.1, workspace:
 pathlib.Path = PosixPath('workspace'), compress_dataset: bool = False,
 random_state: int = 0, sampling_patience: int = 500, **kwargs: Any)
```

Bases: `synthcity.plugins.core.plugin.Plugin`



Bayesian Network for generative modeling. Implemented using pgmpy backend. :param struct\_learning\_n\_iter: int

Number of iterations for the DAG learning

#### Parameters

- **struct\_learning\_search\_method** – str = “tree\_search” Search method for learning the DAG: hillclimb, pc, tree\_search, mmhc, exhaustive
- **struct\_learning\_score** – str = “k2”, Scoring for the DAG search: k2, bdeu, bic, bds
- **struct\_max\_indegree** – int = 4 The maximum number of parents for each node.
- **encoder\_max\_clusters** – int = 10 Data encoding clusters.
- **encoder\_noise\_scale** – float. Small noise to add to the final data, to prevent data leakage.

- **workspace** – Path. Optional Path for caching intermediary results.
- **compress\_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **random\_state** – int. Random seed.
- **sampling\_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

### Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("bayesian_network")
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

### class Config

Bases: object

**arbitrary\_types\_allowed** = True

**validate\_assignment** = True

**fit**(X: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any

Training method the synthetic data plugin.

### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import _
↳ GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
```

(continues on next page)



(continued from previous page)

```

>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "=", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()

```

**Returns** self

**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any) → synthcity.plugins.core.data\_loader.DataLoader

Synthetic data generation method.

#### Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

#### Valid Operations:

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "=", "eq": equal with <value>

- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "==", 0),
>>>]
>>>)
>>>
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**static name**() → str  
The name of the plugin.

**plot**(plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any) → Any  
Plot the real-synthetic distributions.

**Parameters**

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(trial: Any, \*args: Any, \*\*kwargs: Any) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(path: pathlib.Path) → bytes

**schema**() → synthcity.plugins.core.schema.Schema  
The reference schema

**schema\_includes**(*other*: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame]) → bool

Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema**() → synthcity.plugins.core.schema.Schema

The internal schema

**static type**() → str

The type of the plugin.

**static version**() → str

API version

**plugin**

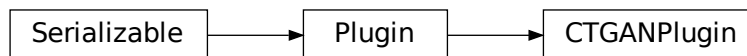
alias of `synthcity.plugins.generic.plugin_bayesian_network.BayesianNetworkPlugin`

## synthcity.plugins.generic.plugin\_ctgan module

Reference: “Modeling Tabular Data using Conditional GAN”, Xu, Lei et al.

**class CTGANPlugin**(*n\_iter*: int = 2000, *generator\_n\_layers\_hidden*: int = 2, *generator\_n\_units\_hidden*: int = 500, *generator\_nonlin*: str = 'relu', *generator\_dropout*: float = 0.1, *generator\_opt\_betas*: tuple = (0.5, 0.999), *discriminator\_n\_layers\_hidden*: int = 2, *discriminator\_n\_units\_hidden*: int = 500, *discriminator\_nonlin*: str = 'leaky\_relu', *discriminator\_n\_iter*: int = 1, *discriminator\_dropout*: float = 0.1, *discriminator\_opt\_betas*: tuple = (0.5, 0.999), *lr*: float = 0.001, *weight\_decay*: float = 0.001, *batch\_size*: int = 200, *random\_state*: int = 0, *clipping\_value*: int = 1, *lambda\_gradient\_penalty*: float = 10, *encoder\_max\_clusters*: int = 10, *encoder*: Any = None, *dataloader\_sampler*: Optional[torch.utils.data.sampler.Sampler] = None, *device*: Any = device(type='cpu'), *patience*: int = 5, *patience\_metric*: Optional[synthcity.metrics.weighted\_metrics.WeightedMetrics] = None, *n\_iter\_print*: int = 50, *n\_iter\_min*: int = 100, *adjust\_inference\_sampling*: bool = False, *workspace*: pathlib.Path = PosixPath('workspace'), *compress\_dataset*: bool = False, *sampling\_patience*: int = 500, *\*\*kwargs*: Any)

Bases: `synthcity.plugins.core.plugin.Plugin`



Conditional Tabular GAN implementation.

### Parameters

- **generator\_n\_layers\_hidden** – int Number of hidden layers in the generator
- **generator\_n\_units\_hidden** – int Number of hidden units in each layer of the Generator
- **generator\_nonlin** – string, default 'leaky\_relu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.

- **n\_iter** – int Maximum number of iterations in the Generator.
- **generator\_dropout** – float Dropout value. If 0, the dropout is not used.
- **discriminator\_n\_layers\_hidden** – int Number of hidden layers in the discriminator
- **discriminator\_n\_units\_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator\_nonlin** – string, default ‘leaky\_relu’ Nonlinearity to use in the discriminator. Can be ‘elu’, ‘relu’, ‘selu’ or ‘leaky\_relu’.
- **discriminator\_n\_iter** – int Maximum number of iterations in the discriminator.
- **discriminator\_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **lr** – float learning rate for optimizer.
- **weight\_decay** – float l2 (ridge) penalty for the weights.
- **batch\_size** – int Batch size
- **random\_state** – int random seed to use
- **clipping\_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **encoder\_max\_clusters** – int The max number of clusters to create for continuous columns when encoding
- **adjust\_inference\_sampling** – bool Adjust the marginal probabilities in the synthetic data to closer match the training set. Active only with the ConditionalSampler
- **stopping** (# *early*) –
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.
- **n\_iter\_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before early stopping is triggered.
- **patience\_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for early stopping.
- **arguments** (# *Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress\_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling\_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

## Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("ctgan", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

### class Config

Bases: object

**arbitrary\_types\_allowed = True**

**validate\_assignment = True**

**fit**(X: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any

Training method the synthetic data plugin.

#### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.data_loader import _
↳ GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
```

(continues on next page)

(continued from previous page)

```

>>> constraints = Constraints(
>>> rules=[
>>> ("target", "=", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()

```

**Returns** self**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any) → synthcity.plugins.core.data\_loader.DataLoader

Synthetic data generation method.

**Parameters**

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

**Valid Operations:**

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "=", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

**Usage example:**

```

>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "=", 0),

```

(continues on next page)

(continued from previous page)

```

>>>]
>>>)
>>>
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()

```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(*\*\*kwargs: Any*) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(*buff: bytes*) → Any

**static load\_dict**(*representation: dict*) → Any

**static name**() → str  
The name of the plugin.

**plot**(*plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any*) → Any  
Plot the real-synthetic distributions.

#### Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(*\*args: Any, \*\*kwargs: Any*) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(*trial: Any, \*args: Any, \*\*kwargs: Any*) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(*path: pathlib.Path*) → bytes

**schema**() → synthcity.plugins.core.schema.Schema  
The reference schema

**schema\_includes**(*other: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame]*) → bool  
Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema**() → synthcity.plugins.core.schema.Schema  
The internal schema

**static type()** → str  
The type of the plugin.

**static version()** → str  
API version

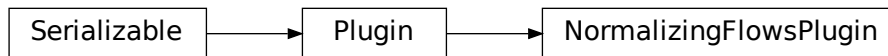
### plugin

alias of `synthcity.plugins.generic.plugin_ctgan.CTGANPlugin`

### synthcity.plugins.generic.plugin\_nflow module

```
class NormalizingFlowsPlugin(n_iter: int = 1000, n_layers_hidden: int = 1, n_units_hidden: int = 100,
 batch_size: int = 200, num_transform_blocks: int = 1, dropout: float = 0.1,
 batch_norm: bool = False, num_bins: int = 8, tail_bound: float = 3, lr: float =
 0.001, apply_unconditional_transform: bool = True, base_distribution: str =
 'standard_normal', linear_transform_type: str = 'permutation',
 base_transform_type: str = 'rq-autoregressive', encoder_max_clusters: int =
 10, tabular: bool = True, n_iter_min: int = 100, n_iter_print: int = 50,
 patience: int = 5, patience_metric:
 Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None,
 workspace: pathlib.Path = PosixPath('workspace'), compress_dataset: bool =
 False, sampling_patience: int = 500, random_state: int = 0, device: Any =
 device(type='cpu'), **kwargs: Any)
```

Bases: `synthcity.plugins.core.plugin.Plugin`



Normalizing Flows methods.

Normalizing Flows are generative models which produce tractable distributions where both sampling and density evaluation can be efficient and exact.

#### Parameters

- **n\_iter** – int Number of flow steps
- **n\_layers\_hidden** – int Number of transformation layers
- **n\_units\_hidden** – int Number of hidden units for each layer
- **batch\_size** – int Size of batch used for training
- **num\_transform\_blocks** – int Number of blocks to use in coupling/autoregressive nets.
- **dropout** – float Dropout probability for coupling/autoregressive nets.
- **batch\_norm** – bool Whether to use batch norm in coupling/autoregressive nets.
- **num\_bins** – int Number of bins to use for piecewise transforms.
- **tail\_bound** – float Box is on  $[-bound, bound]^2$
- **lr** – float Learning rate for optimizer.



- **apply\_unconditional\_transform** – bool Whether to unconditionally transform ‘identity’ features in the coupling layer.
- **base\_distribution** – str Possible values: “standard\_normal”
- **linear\_transform\_type** – str Type of linear transform to use. Possible values:
  - lu : A linear transform where we parameterize the LU decomposition of the weights.
  - permutation: Permutes using a random, but fixed, permutation.
  - svd: A linear module using the SVD decomposition for the weight matrix.
- **base\_transform\_type** – str Type of transform to use between linear layers. Possible values:
  - **affine-coupling** [An affine coupling layer that scales and shifts part of the variables.] Ref: L. Dinh et al., “Density estimation using Real NVP”.
  - **quadratic-coupling** : Ref: Müller et al., “Neural Importance Sampling”.
  - **rq-coupling** [Rational Quadratic Coupling] Ref: Durkan et al, “Neural Spline Flows”.
  - **affine-autoregressive :Affine Autoregressive Transform** Ref: Durkan et al, “Neural Spline Flows”.
  - **quadratic-autoregressive** [Quadratic Autoregressive Transform] Ref: Durkan et al, “Neural Spline Flows”.
  - **rq-autoregressive** [Rational Quadratic Autoregressive Transform] Ref: Durkan et al, “Neural Spline Flows”.
- **stopping** (# *early*) –
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.
- **n\_iter\_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before training early stopping is trigged.
- **patience\_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for training early stopping.
- **arguments** (# *Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress\_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling\_patience** – int. Max inference iterations to wait for the generated data to match the training schema.
- **random\_state** – int random seed to use

### Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("nflow", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

### class Config

Bases: object

**arbitrary\_types\_allowed = True**

**validate\_assignment = True**

**fit**(X: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any

Training method the synthetic data plugin.

### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.data_loader import _
↳ GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
```

(continues on next page)

(continued from previous page)

```

>>> constraints = Constraints(
>>> rules=[
>>> ("target", "=", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()

```

**Returns** self**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(*count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any*) → [synthcity.plugins.core.data\\_loader.DataLoader](#)

Synthetic data generation method.

**Parameters**

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

**Valid Operations:**

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "=", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

**Usage example:**

```

>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "=", 0),

```

(continues on next page)

(continued from previous page)

```

>>>]
>>>)
>>>
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()

```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(*\*\*kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]  
Returns the hyperparameter space for the derived plugin.

**static load**(*buff: bytes*) → Any

**static load\_dict**(*representation: dict*) → Any

**static name**() → str  
The name of the plugin.

**plot**(*plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any*) → Any  
Plot the real-synthetic distributions.

#### Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(*\*args: Any, \*\*kwargs: Any*) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(*trial: Any, \*args: Any, \*\*kwargs: Any*) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(*path: pathlib.Path*) → bytes

**schema**() → *synthcity.plugins.core.schema.Schema*  
The reference schema

**schema\_includes**(*other: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame]*) → bool  
Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema**() → *synthcity.plugins.core.schema.Schema*  
The internal schema

**static type()** → str  
The type of the plugin.

**static version()** → str  
API version

### plugin

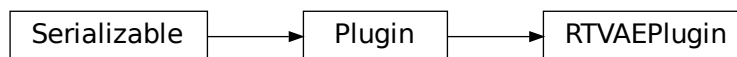
alias of `synthcity.plugins.generic.plugin_nflow.NormalizingFlowsPlugin`

## synthcity.plugins.generic.plugin\_rtvae module

Reference: Akrami, Haleh, Anand A. Joshi, Jian Li, Sergül Aydıno, and Richard M. Leahy. “A robust variational autoencoder using beta divergence.” Knowledge-Based Systems 238 (2022): 107886.

```
class RTVAEPlugin(n_iter: int = 1000, n_units_embedding: int = 500, lr: float = 0.001, weight_decay: float = 1e-05, batch_size: int = 200, random_state: int = 0, decoder_n_layers_hidden: int = 3, decoder_n_units_hidden: int = 500, decoder_nonlin: str = 'leaky_relu', decoder_dropout: float = 0, encoder_n_layers_hidden: int = 3, encoder_n_units_hidden: int = 500, encoder_nonlin: str = 'leaky_relu', encoder_dropout: float = 0.1, data_encoder_max_clusters: int = 10, robust_divergence_beta: int = 2, dataloader_sampler: Optional[torch.utils.data.sampler.Sampler] = None, n_iter_print: int = 50, n_iter_min: int = 100, patience: int = 5, device: Any = device(type='cpu'), workspace: pathlib.Path = PosixPath('workspace'), compress_dataset: bool = False, sampling_patience: int = 500, **kwargs: Any)
```

Bases: `synthcity.plugins.core.plugin.Plugin`



Tabular VAE with robust beta divergence.

### Parameters

- **decoder\_n\_layers\_hidden** – int Number of hidden layers in the decoder
- **decoder\_n\_units\_hidden** – int Number of hidden units in each layer of the decoder
- **decoder\_nonlin** – string, default ‘leaky\_relu’ Nonlinearity to use in the decoder. Can be ‘elu’, ‘relu’, ‘selu’ or ‘leaky\_relu’.
- **decoder\_dropout** – float Dropout value. If 0, the dropout is not used.
- **encoder\_n\_layers\_hidden** – int Number of hidden layers in the encoder
- **encoder\_n\_units\_hidden** – int Number of hidden units in each layer of the encoder
- **encoder\_nonlin** – string, default ‘leaky\_relu’ Nonlinearity to use in the encoder. Can be ‘elu’, ‘relu’, ‘selu’ or ‘leaky\_relu’.
- **encoder\_dropout** – float Dropout value for the encoder. If 0, the dropout is not used.
- **n\_iter** – int Maximum number of iterations in the encoder.
- **lr** – float learning rate for optimizer.

- **weight\_decay** – float l2 (ridge) penalty for the weights.
- **batch\_size** – int Batch size
- **random\_state** – int random\_state used
- **encoder\_max\_clusters** – int The max number of clusters to create for continuous columns when encoding
- **arguments** (# Core Plugin) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress\_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling\_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

### Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("rtvae", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

### class Config

Bases: object

**arbitrary\_types\_allowed** = True

**validate\_assignment** = True

**fit**(X: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any

Training method the synthetic data plugin.

### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.data_loader import _
↳ GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
```

(continues on next page)

(continued from previous page)

```

>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "=", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()

```

**Returns** self**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any) → synthcity.plugins.core.data\_loader.DataLoader

Synthetic data generation method.

**Parameters**

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

**Valid Operations:**

- "<" , "<=" : less than <value>

- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

**Usage example:**

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "==", 0),
>>>]
>>>)
>>>
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**static name**() → str  
The name of the plugin.

**plot**(plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any) → Any  
Plot the real-synthetic distributions.

**Parameters**

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(trial: Any, \*args: Any, \*\*kwargs: Any) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict



**save\_to\_file**(*path*: *pathlib.Path*) → bytes

**schema**() → *synthcity.plugins.core.schema.Schema*  
The reference schema

**schema\_includes**(*other*: *Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame]*) → bool  
Helper method to test if the reference schema includes a Dataset

**Parameters** *other* – *DataLoader*. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema**() → *synthcity.plugins.core.schema.Schema*  
The internal schema

**static type**() → str  
The type of the plugin.

**static version**() → str  
API version

**plugin**

alias of *synthcity.plugins.generic.plugin\_rtvae.RTVAEPlugin*

**synthcity.plugins.generic.plugin\_tvae module**

**class TVAEPlugin**(*n\_iter*: int = 1000, *n\_units\_embedding*: int = 500, *lr*: float = 0.001, *weight\_decay*: float = 1e-05, *batch\_size*: int = 200, *random\_state*: int = 0, *decoder\_n\_layers\_hidden*: int = 3, *decoder\_n\_units\_hidden*: int = 500, *decoder\_nonlin*: str = 'leaky\_relu', *decoder\_dropout*: float = 0, *encoder\_n\_layers\_hidden*: int = 3, *encoder\_n\_units\_hidden*: int = 500, *encoder\_nonlin*: str = 'leaky\_relu', *encoder\_dropout*: float = 0.1, *loss\_factor*: int = 1, *data\_encoder\_max\_clusters*: int = 10, *data\_loader\_sampler*: *Optional[torch.utils.data.sampler.Sampler]* = None, *clipping\_value*: int = 1, *n\_iter\_print*: int = 50, *n\_iter\_min*: int = 100, *patience*: int = 5, *device*: Any = *device(type='cpu')*, *workspace*: *pathlib.Path* = *PosixPath('workspace')*, *compress\_dataset*: bool = False, *sampling\_patience*: int = 500, *\*\*kwargs*: Any)

Bases: *synthcity.plugins.core.plugin.Plugin*



Tabular VAE implementation.

**Parameters**

- **decoder\_n\_layers\_hidden** – int Number of hidden layers in the decoder
- **decoder\_n\_units\_hidden** – int Number of hidden units in each layer of the decoder
- **decoder\_nonlin** – string, default 'leaky\_relu' Nonlinearity to use in the decoder. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **decoder\_dropout** – float Dropout value. If 0, the dropout is not used.

- **encoder\_n\_layers\_hidden** – int Number of hidden layers in the encoder
- **encoder\_n\_units\_hidden** – int Number of hidden units in each layer of the encoder
- **encoder\_nonlin** – string, default 'leaky\_relu' Nonlinearity to use in the encoder. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **encoder\_dropout** – float Dropout value for the encoder. If 0, the dropout is not used.
- **n\_iter** – int Maximum number of iterations in the encoder.
- **lr** – float learning rate for optimizer.
- **weight\_decay** – float l2 (ridge) penalty for the weights.
- **batch\_size** – int Batch size
- **random\_state** – int random\_state used
- **encoder\_max\_clusters** – int The max number of clusters to create for continuous columns when encoding
- **stopping** (# *early*) –
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.
- **n\_iter\_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before early stopping is triggered.
- **arguments** (# *Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress\_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling\_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

### Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("tvae", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

```
class Config
 Bases: object

 arbitrary_types_allowed = True
 validate_assignment = True
```

**fit**(*X*: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any  
 Training method the synthetic data plugin.

#### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.data_loader import _
↳GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "==", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()
```

**Returns** self

**classmethod fqdn**() → str

The Fully-Qualified name of the plugin.

**generate**(count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any) →  
 synthcity.plugins.core.data\_loader.DataLoader

Synthetic data generation method.

### Parameters

- **count** – optional int. The number of samples to generate. If None, it generated `len(reference_dataset)` samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

#### Valid Operations:

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

#### Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "==", 0),
>>>]
>>>)
>>>
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]**  
Returns the hyperparameter space for the derived plugin.

**static load(buff: bytes) → Any**

**static load\_dict(representation: dict) → Any**

**static name() → str**  
The name of the plugin.

**plot**(*plt*: Any, *X*: [synthcity.plugins.core.dataloader.DataLoader](#), *count*: *Optional[int]* = None, *plots*: list = ['marginal', 'associations', 'tsne'], *\*\*kwargs*: Any) → Any  
 Plot the real-synthetic distributions.

#### Parameters

- **plt** – output
- **X** – [DataLoader](#). The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(*\*args*: Any, *\*\*kwargs*: Any) → Dict[str, Any]  
 Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(*trial*: Any, *\*args*: Any, *\*\*kwargs*: Any) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(*path*: [pathlib.Path](#)) → bytes

**schema**() → [synthcity.plugins.core.schema.Schema](#)  
 The reference schema

**schema\_includes**(*other*: *Union*[[synthcity.plugins.core.dataloader.DataLoader](#), [pandas.core.frame.DataFrame](#)]) → bool  
 Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – [DataLoader](#). The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema**() → [synthcity.plugins.core.schema.Schema](#)  
 The internal schema

**static type**() → str  
 The type of the plugin.

**static version**() → str  
 API version

#### plugin

alias of [synthcity.plugins.generic.plugin\\_tvae.TVAEPlugin](#)

## 10.1.2 Privacy-focused

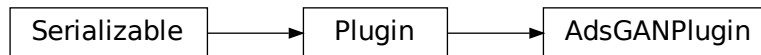
### [synthcity.plugins.privacy.plugin\\_adsgan](#) module

**Reference:** Jinsung Yoon, Lydia N. Drumright, Mihaela van der Schaar, “Anonymization through Data Synthesis using Generative Adversarial Networks (ADS-GAN): A harmonizing advancement for AI in medicine,” IEEE Journal of Biomedical and Health Informatics (JBHI), 2019.

Paper link: <https://ieeexplore.ieee.org/document/9034117>

```
class AdsGANPlugin(n_iter: int = 10000, generator_n_layers_hidden: int = 2, generator_n_units_hidden: int =
 500, generator_nonlin: str = 'relu', generator_dropout: float = 0.1, generator_opt_betas:
 tuple = (0.5, 0.999), discriminator_n_layers_hidden: int = 2,
 discriminator_n_units_hidden: int = 500, discriminator_nonlin: str = 'leaky_relu',
 discriminator_n_iter: int = 1, discriminator_dropout: float = 0.1, discriminator_opt_betas:
 tuple = (0.5, 0.999), lr: float = 0.001, weight_decay: float = 0.001, batch_size: int = 200,
 random_state: int = 0, clipping_value: int = 1, lambda_gradient_penalty: float = 10,
 lambda_identifiability_penalty: float = 0.1, encoder_max_clusters: int = 5, encoder: Any =
 None, dataloader_sampler: Optional[torch.utils.data.sampler.Sampler] = None, device:
 Any = device(type='cpu'), adjust_inference_sampling: bool = False, patience: int = 5,
 patience_metric: Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None,
 n_iter_print: int = 50, n_iter_min: int = 100, workspace: pathlib.Path =
 PosixPath('workspace'), compress_dataset: bool = False, sampling_patience: int = 500,
 **kwargs: Any)
```

Bases: [synthcity.plugins.core.plugin.Plugin](#)



AdsGAN plugin - Anonymization through Data Synthesis using Generative Adversarial Networks.

#### Parameters

- **generator\_n\_layers\_hidden** – int Number of hidden layers in the generator
- **generator\_n\_units\_hidden** – int Number of hidden units in each layer of the Generator
- **generator\_nonlin** – string, default 'leaky\_relu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **n\_iter** – int Maximum number of iterations in the Generator.
- **generator\_dropout** – float Dropout value. If 0, the dropout is not used.
- **discriminator\_n\_layers\_hidden** – int Number of hidden layers in the discriminator
- **discriminator\_n\_units\_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator\_nonlin** – string, default 'leaky\_relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **discriminator\_n\_iter** – int Maximum number of iterations in the discriminator.
- **discriminator\_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **lr** – float learning rate for optimizer.
- **weight\_decay** – float l2 (ridge) penalty for the weights.
- **batch\_size** – int Batch size
- **random\_state** – int random seed to use

- **clipping\_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **encoder\_max\_clusters** – int The max number of clusters to create for continuous columns when encoding
- **adjust\_inference\_sampling** – bool Adjust the marginal probabilities in the synthetic data to closer match the training set. Active only with the ConditionalSampler
- **lambda\_gradient\_penalty** – float = 10 Weight for the gradient penalty
- **lambda\_identifiability\_penalty** – float = 0.1 Weight for the identifiability penalty, if enabled
- **stopping** (# *early*) –
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.
- **n\_iter\_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before training early stopping is triggered.
- **patience\_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for training early stopping.
- **arguments** (# *Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress\_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling\_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

### Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("adsgan", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

```
class Config
```

```
 Bases: object
```

```
 arbitrary_types_allowed = True
```

```
 validate_assignment = True
```

```
fit(X: Union[synthcity.plugins.core.data_loader.DataLoader, pandas.core.frame.DataFrame], *args: Any,
 **kwargs: Any) → Any
```

```
 Training method the synthetic data plugin.
```

**Parameters**

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import _
GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "==", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()
```

**Returns** self

**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any) → synthcity.plugins.core.dataloader.DataLoader

Synthetic data generation method.

**Parameters**

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.



- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

#### Valid Operations:

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

#### Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "==", 0),
>>>]
>>>)
>>>
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**static name**() → str  
The name of the plugin.

**plot**(plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any) → Any  
Plot the real-synthetic distributions.

#### Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]

Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(trial: Any, \*args: Any, \*\*kwargs: Any) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(path: pathlib.Path) → bytes

**schema**() → *synthcity.plugins.core.schema.Schema*  
The reference schema

**schema\_includes**(other: Union[*synthcity.plugins.core.data\_loader.DataLoader*,  
*pandas.core.frame.DataFrame*]) → bool  
Helper method to test if the reference schema includes a Dataset

**Parameters other** – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema**() → *synthcity.plugins.core.schema.Schema*  
The internal schema

**static type**() → str  
The type of the plugin.

**static version**() → str  
API version

### plugin

alias of *synthcity.plugins.privacy.plugin\_adsgan.AdsGANPlugin*

## synthcity.plugins.privacy.plugin\_pategan module

Reference: James Jordon, Jinsung Yoon, Mihaela van der Schaar, “PATE-GAN: Generating Synthetic Data with Differential Privacy Guarantees,” International Conference on Learning Representations (ICLR), 2019. Paper link: <https://openreview.net/forum?id=S1zk9iRqF7>

**class PATEGAN**(max\_iter: int = 1000, generator\_n\_layers\_hidden: int = 2, generator\_n\_units\_hidden: int = 100, generator\_nonlin: str = 'relu', generator\_n\_iter: int = 10, generator\_dropout: float = 0, discriminator\_n\_layers\_hidden: int = 2, discriminator\_n\_units\_hidden: int = 100, discriminator\_nonlin: str = 'leaky\_relu', discriminator\_n\_iter: int = 1, discriminator\_dropout: float = 0.1, lr: float = 0.0001, weight\_decay: float = 0.001, batch\_size: int = 200, random\_state: int = 0, clipping\_value: int = 1, encoder\_max\_clusters: int = 5, device: Any = device(type='cpu'), n\_teachers: int = 10, teacher\_template: str = 'linear', epsilon: float = 1.0, delta: Optional[float] = None, lamda: float = 0.001, alpha: int = 100, encoder: Any = None)

Bases: *synthcity.plugins.core.serializable.Serializable*

Basic PATE-GAN framework.

**fit**(X\_train: *pandas.core.frame.DataFrame*) → Any

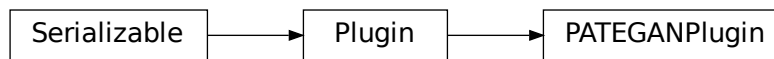
**static load**(buff: bytes) → Any

```

static load_dict(representation: dict) → Any
sample(count: int) → numpy.ndarray
save() → bytes
save_dict() → dict
save_to_file(path: pathlib.Path) → bytes
static version() → str
 API version
class PATEGANPlugin(n_iter: int = 200, generator_n_iter: int = 10, generator_n_layers_hidden: int = 2,
 generator_n_units_hidden: int = 500, generator_nonlin: str = 'relu', generator_dropout:
 float = 0, discriminator_n_layers_hidden: int = 2, discriminator_n_units_hidden: int =
 500, discriminator_nonlin: str = 'leaky_relu', discriminator_n_iter: int = 1,
 discriminator_dropout: float = 0.1, lr: float = 0.001, weight_decay: float = 0.001,
 batch_size: int = 200, random_state: int = 0, clipping_value: int = 1,
 encoder_max_clusters: int = 5, n_teachers: int = 10, teacher_template: str = 'xgboost',
 epsilon: float = 1.0, delta: Optional[float] = None, lamda: float = 0.001, alpha: int = 100,
 encoder: Any = None, device: Any = device(type='cpu'), workspace: pathlib.Path =
 PosixPath('workspace'), compress_dataset: bool = False, sampling_patience: int = 500,
 **kwargs: Any)

```

Bases: [synthcity.plugins.core.plugin.Plugin](#)



PATE-GAN: Generating Synthetic Data with Differential Privacy Guarantees.

#### Parameters

- **generator\_n\_layers\_hidden** – int Number of hidden layers in the generator
- **generator\_n\_units\_hidden** – int Number of hidden units in each layer of the Generator
- **generator\_nonlin** – string, default 'leaky\_relu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **n\_iter** – int Maximum number of iterations in the Generator.
- **generator\_dropout** – float Dropout value. If 0, the dropout is not used.
- **discriminator\_n\_layers\_hidden** – int Number of hidden layers in the discriminator
- **discriminator\_n\_units\_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator\_nonlin** – string, default 'leaky\_relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **discriminator\_n\_iter** – int Maximum number of iterations in the discriminator.
- **discriminator\_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.

- **lr** – float learning rate for optimizer.
- **weight\_decay** – float l2 (ridge) penalty for the weights.
- **batch\_size** – int Batch size
- **random\_state** – int random\_state used
- **clipping\_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **teacher\_template** – str Model to use for the teachers. Can be linear, xgboost.
- **epsilon** – float Differential privacy parameter
- **delta** – float Differential privacy parameter
- **lambda** – float Noise size
- **encoder\_max\_clusters** – int The max number of clusters to create for continuous columns when encoding
- **arguments** (# Core Plugin) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress\_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling\_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

### Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("pategan", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

#### class Config

Bases: object

**arbitrary\_types\_allowed = True**

**validate\_assignment = True**

**fit**(X: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any

Training method the synthetic data plugin.

#### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like

GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.data_loader import _
↳ GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "==", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()
```

Returns self

**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(count: *Optional[int] = None*, constraints: *Optional[synthcity.plugins.core.constraints.Constraints] = None*, random\_state: *Optional[int] = None*, \*\*kwargs: *Any*) → *synthcity.plugins.core.data\_loader.DataLoader*

Synthetic data generation method.

#### Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.

- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

**Valid Operations:**

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

**Usage example:**

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "=", 0),
>>>]
>>>)
>>>
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]

Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**static name**() → str

The name of the plugin.

**plot**(plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any) → Any

Plot the real-synthetic distributions.

**Parameters**

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

---

```

classmethod sample_hyperparameters(*args: Any, **kwargs: Any) → Dict[str, Any]
 Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(trial: Any, *args: Any, **kwargs: Any) → Dict[str,
 Any]

save() → bytes

save_dict() → dict

save_to_file(path: pathlib.Path) → bytes

schema() → synthcity.plugins.core.schema.Schema
 The reference schema

schema_includes(other: Union[synthcity.plugins.core.data_loader.DataLoader,
 pandas.core.frame.DataFrame]) → bool
 Helper method to test if the reference schema includes a Dataset

 Parameters other – DataLoader. The dataset to test

 Returns bool, if the schema includes the dataset or not.

training_schema() → synthcity.plugins.core.schema.Schema
 The internal schema

static type() → str
 The type of the plugin.

static version() → str
 API version

class Teachers(n_teachers: int, samples_per_teacher: int, lamda: float = 0.001, template: str = 'xgboost')
 Bases: synthcity.plugins.core.serializable.Serializable

 fit(X: numpy.ndarray, generator: Any) → Any

 static load(buff: bytes) → Any

 static load_dict(representation: dict) → Any

 pate_lambda(x: numpy.ndarray) → Tuple[int, int, int]
 Returns PATE_lambda(x).

 Parameters x (-) – feature vector

 Returns the number of label 0 and 1, respectively - out: label after adding laplace noise.

 Return type
 • n0, n1

 save() → bytes

 save_dict() → dict

 save_to_file(path: pathlib.Path) → bytes

 static version() → str
 API version

plugin
 alias of synthcity.plugins.privacy.plugin_pategan.PATEGANPlugin

```

## synthcity.plugins.privacy.plugin\_privbayes module

Reference: PrivBayes: Private Data Release via Bayesian Networks. (2017), Zhang J, Cormode G, Procopiuc CM, Srivastava D, Xiao X.

**class PrivBayes**(*epsilon: float = 1.0, K: int = 0, n\_bins: int = 100, mi\_thresh: float = 0.01, target\_usefulness: int = 5*)

Bases: [synthcity.plugins.core.serializable.Serializable](#)

PrivBayes is a differentially private method for releasing high-dimensional data.

**Given a dataset D, PrivBayes first constructs a Bayesian network N , which**

- (i) provides a succinct model of the correlations among the attributes in D
- (ii) allows us to approximate the distribution of data in D using a set P of lowdimensional marginals of D.

After that, PrivBayes injects noise into each marginal in P to ensure differential privacy, and then uses the noisy marginals and the Bayesian network to construct an approximation of the data distribution in D. Finally, PrivBayes samples tuples from the approximate distribution to construct a synthetic dataset, and then releases the synthetic data.

**display\_network**() → None

**fit**(*data: pandas.core.frame.DataFrame*) → Any

**static load**(*buff: bytes*) → Any

**static load\_dict**(*representation: dict*) → Any

**mutual\_info\_score**(*data: pandas.core.frame.DataFrame, parents: List[str], candidate: str*) → float  
Cluster the source columns, and compute the mutual information between the target and the clusters.

**sample**(*count: int*) → pandas.core.frame.DataFrame

**save**() → bytes

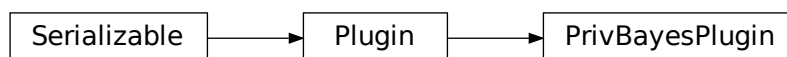
**save\_dict**() → dict

**save\_to\_file**(*path: pathlib.Path*) → bytes

**static version**() → str  
API version

**class PrivBayesPlugin**(*epsilon: float = 1.0, K: int = 0, n\_bins: int = 100, mi\_thresh: float = 0.01, target\_usefulness: int = 5, random\_state: int = 0, workspace: pathlib.Path = PosixPath('workspace'), compress\_dataset: bool = False, sampling\_patience: int = 500, \*\*kwargs: Any*)

Bases: [synthcity.plugins.core.plugin.Plugin](#)



PrivBayes algorithm.

**Args:**



**epsilon: float** Differential privacy parameter

**K:** Maximum number of parents for a node

**n\_bins: int** Number of bins for encoding the features

**mi\_thresh: int** Mutual information lower threshold. If the current score is lower, the [] parents are used.

**target\_usefulness: int** Def 4.7 in the paper: A noisy distribution is -useful if the ratio of average scale of

**information to average scale of noise is no less than . 5-useful is the recommended value.**

**random\_state: int** Random seed

# Core Plugin arguments workspace: Path.

Optional Path for caching intermediary results.

**compress\_dataset: bool. Default = False.** Drop redundant features before training the generator.

**sampling\_patience: int.** Max inference iterations to wait for the generated data to match the training schema.

## Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("privbayes")
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

**class Config**

Bases: object

**arbitrary\_types\_allowed = True**

**validate\_assignment = True**

**fit**(X: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any

Training method the synthetic data plugin.

### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```

>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.data_loader import _
↳ GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "==", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()

```

Returns self

**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any) → synthcity.plugins.core.data\_loader.DataLoader

Synthetic data generation method.

#### Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated

data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

#### Valid Operations:

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

#### Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "=", 0),
>>>]
>>>)
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**static name**() → str  
The name of the plugin.

**plot**(plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any) → Any  
Plot the real-synthetic distributions.

#### Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod** **sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**classmethod** **sample\_hyperparameters\_optuna**(trial: Any, \*args: Any, \*\*kwargs: Any) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(path: pathlib.Path) → bytes

**schema**() → [synthcity.plugins.core.schema.Schema](#)  
The reference schema

**schema\_includes**(other: Union[[synthcity.plugins.core.dataloader.DataLoader](#), [pandas.core.frame.DataFrame](#)]) → bool  
Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema**() → [synthcity.plugins.core.schema.Schema](#)  
The internal schema

**static type**() → str  
The type of the plugin.

**static version**() → str  
API version

**class** **network\_edge**(feature, parents)  
Bases: tuple

**count**(value, /)  
Return number of occurrences of value.

**feature**  
Alias for field number 0

**index**(value, start=0, stop=9223372036854775807, /)  
Return first index of value.  
  
Raises ValueError if the value is not present.

**parents**  
Alias for field number 1

**plugin**  
alias of [synthcity.plugins.privacy.plugin\\_privbayes.PrivBayesPlugin](#)

**usefulness\_minus\_target**(k: int, num\_attributes: int, num\_tuples: int, target\_usefulness: int = 5, epsilon: float = 0.1) → int  
Usefulness function in PrivBayes.

**Parameters**

- **k** (int) – Max number of degree in Bayesian networks construction
- **num\_attributes** (int) – Number of attributes in dataset.
- **num\_tuples** (int) – Number of tuples in dataset.
- **target\_usefulness** (int or float) –

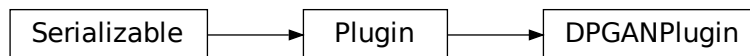
- **epsilon** (*float*) – Parameter of differential privacy.

### synthcity.plugins.privacy.plugin\_dpgan module

Reference: “Differentially Private Generative Adversarial Network”, Xie, Liyang et al.

```
class DPGANPlugin(n_iter: int = 2000, generator_n_layers_hidden: int = 2, generator_n_units_hidden: int =
 500, generator_nonlin: str = 'relu', generator_dropout: float = 0.1, generator_opt_betas:
 tuple = (0.5, 0.999), discriminator_n_layers_hidden: int = 2, discriminator_n_units_hidden:
 int = 500, discriminator_nonlin: str = 'leaky_relu', discriminator_n_iter: int = 1,
 discriminator_dropout: float = 0.1, discriminator_opt_betas: tuple = (0.5, 0.999), lr: float =
 0.001, weight_decay: float = 0.001, batch_size: int = 200, random_state: int = 0,
 clipping_value: int = 1, lambda_gradient_penalty: float = 10, encoder_max_clusters: int =
 5, encoder: Any = None, dataloader_sampler: Optional[torch.utils.data.sampler.Sampler] =
 None, device: Any = device(type='cpu'), epsilon: float = 1, delta: Optional[float] = None,
 dp_max_grad_norm: float = 2, dp_secure_mode: bool = False, patience: int = 5,
 patience_metric: Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None,
 n_iter_print: int = 50, n_iter_min: int = 100, workspace: pathlib.Path =
 PosixPath('workspace'), compress_dataset: bool = False, sampling_patience: int = 500,
 **kwargs: Any)
```

Bases: `synthcity.plugins.core.plugin.Plugin`



Differentially Private Generative Adversarial Network implementation. The discriminator is trained using DP-SGD.

#### Parameters

- **generator\_n\_layers\_hidden** – *int* Number of hidden layers in the generator
- **generator\_n\_units\_hidden** – *int* Number of hidden units in each layer of the Generator
- **generator\_nonlin** – *string*, default 'leaky\_relu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **n\_iter** – *int* Maximum number of iterations in the Generator.
- **generator\_dropout** – *float* Dropout value. If 0, the dropout is not used.
- **discriminator\_n\_layers\_hidden** – *int* Number of hidden layers in the discriminator
- **discriminator\_n\_units\_hidden** – *int* Number of hidden units in each layer of the discriminator
- **discriminator\_nonlin** – *string*, default 'leaky\_relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **discriminator\_n\_iter** – *int* Maximum number of iterations in the discriminator.
- **discriminator\_dropout** – *float* Dropout value for the discriminator. If 0, the dropout is not used.

- **lr** – float learning rate for optimizer.
- **weight\_decay** – float l2 (ridge) penalty for the weights.
- **batch\_size** – int Batch size
- **random\_state** – int random seed to use
- **clipping\_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **encoder\_max\_clusters** – int The max number of clusters to create for continuous columns when encoding
- **stopping** (# *early*) –
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.
- **n\_iter\_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before early stopping is triggered.
- **patience\_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for early stopping.
- **settings** (# *privacy*) –
- **dp\_enabled** – bool Train the discriminator with Differential Privacy guarantees
- **dp\_delta** – Optional[float] Optional DP delta: the probability of information accidentally being leaked. Usually 1 / len(dataset)
- **dp\_epsilon** – float = 3 DP epsilon: privacy budget, which is a measure of the amount of privacy that is preserved by a given algorithm. Epsilon is a number that represents the maximum amount of information that an adversary can learn about an individual from the output of a differentially private algorithm. The smaller the value of epsilon, the more private the algorithm is. For example, an algorithm with an epsilon of 0.1 preserves more privacy than an algorithm with an epsilon of 1.0.
- **dp\_max\_grad\_norm** – float max grad norm used for gradient clipping
- **dp\_secure\_mode** – bool = False, if True uses noise generation approach robust to floating point arithmetic attacks.

### Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("dpgan", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

Note: There is a known issue with the training step for training GANs with conditionals with `dp_enabled` set to True, as is the case for DPGAN.

```
class Config
```

```
 Bases: object
```

```
 arbitrary_types_allowed = True
```

```
 validate_assignment = True
```

```
fit(X: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame], *args: Any,
 **kwargs: Any) → Any
```

Training method the synthetic data plugin.

#### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import _
 _GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "==", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()
```

Returns self

```
classmethod fqdn() → str
```

The Fully-Qualified name of the plugin.

**generate**(*count*: *Optional[int] = None*, *constraints*: *Optional[synthcity.plugins.core.constraints.Constraints]* = *None*, *random\_state*: *Optional[int] = None*, *\*\*kwargs*: *Any*) → *synthcity.plugins.core.data\_loader.DataLoader*

Synthetic data generation method.

#### Parameters

- **count** – optional int. The number of samples to generate. If None, it generated `len(reference_dataset)` samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

#### Valid Operations:

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

#### Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "=", 0),
>>>]
>>>)
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(*\*\*kwargs*: *Any*) → List[*synthcity.plugins.core.distribution.Distribution*]

Returns the hyperparameter space for the derived plugin.

**static load**(*buff*: *bytes*) → *Any*



**static load\_dict**(*representation: dict*) → Any

**static name**() → str

The name of the plugin.

**plot**(*plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any*) → Any

Plot the real-synthetic distributions.

#### Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]

Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(*trial: Any, \*args: Any, \*\*kwargs: Any*) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(*path: pathlib.Path*) → bytes

**schema**() → *synthcity.plugins.core.schema.Schema*

The reference schema

**schema\_includes**(*other: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame]*) → bool

Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema**() → *synthcity.plugins.core.schema.Schema*

The internal schema

**static type**() → str

The type of the plugin.

**static version**() → str

API version

#### plugin

alias of *synthcity.plugins.privacy.plugin\_dpghan.DPGANPlugin*

### synthcity.plugins.privacy.plugin\_decaf module

Reference: Boris van Breugel, Trent Kyono, Jeroen Berrevoets, Mihaela van der Schaar “DECAF: Generating Fair Synthetic Data Using Causally-Aware Generative Networks”(2021).

```
class DECAFPlugin(n_iter: int = 100, n_iter_baseline: int = 1000, generator_n_layers_hidden: int = 2,
 generator_n_units_hidden: int = 500, generator_nonlin: str = 'relu', generator_dropout:
 float = 0.1, generator_opt_betas: tuple = (0.5, 0.999), discriminator_n_layers_hidden: int =
 2, discriminator_n_units_hidden: int = 500, discriminator_nonlin: str = 'leaky_relu',
 discriminator_n_iter: int = 1, discriminator_dropout: float = 0.1, discriminator_opt_betas:
 tuple = (0.5, 0.999), lr: float = 0.001, batch_size: int = 200, random_state: int = 0,
 clipping_value: int = 1, lambda_gradient_penalty: float = 10, lambda_privacy: float = 1,
 eps: float = 1e-08, alpha: float = 1, rho: float = 1, weight_decay: float = 0.01, ll_g: float =
 0, ll_W: float = 1, grad_dag_loss: bool = False, struct_learning_enabled: bool = True,
 struct_learning_n_iter: int = 1000, struct_learning_search_method: str = 'tree_search',
 struct_learning_score: str = 'k2', struct_max_indegree: int = 4, encoder_max_clusters: int =
 10, device: Any = device(type='cpu'), workspace: pathlib.Path = PosixPath('workspace'),
 compress_dataset: bool = False, sampling_patience: int = 500, **kwargs: Any)
```

Bases: [synthcity.plugins.core.plugin.Plugin](#)



DECAF (DEbiasing CAusal Fairness) plugin.

#### Parameters

- **n\_iter** – int Number of training iterations.
- **generator\_n\_layers\_hidden** – int Number of hidden layers in the generator.
- **generator\_n\_units\_hidden** – Number of neurons in the hidden layers of the generator.
- **generator\_nonlin** – str Nonlinearity used by the generator for the hidden layers: leaky\_relu, relu, gelu etc.
- **generator\_dropout** – float Generator dropout.
- **generator\_opt\_betas** – tuple Generator initial decay rates for the Adam optimizer
- **discriminator\_n\_layers\_hidden** – int Number of hidden layers in the discriminator.
- **discriminator\_n\_units\_hidden** – int Number of neurons in the hidden layers of the discriminator.
- **discriminator\_nonlin** – str Nonlinearity used by the discriminator for the hidden layers: leaky\_relu, relu, gelu etc.
- **discriminator\_n\_iter** – int Discriminator number of iterations(default = 1)
- **discriminator\_dropout** – float Discriminator dropout
- **discriminator\_opt\_betas** – tuple Discriminator initial decay rates for the Adam optimizer
- **lr** – float Learning rate
- **weight\_decay** – float Optimizer weight decay
- **batch\_size** – int Batch size
- **random\_state** – int Random seed

- **clipping\_value** – int Gradient clipping value
- **lambda\_gradient\_penalty** – float Gradient penalty factor used for training the GAN.
- **lambda\_privacy** – float Privacy factor used the AdsGAN loss.
- **eps** – float = 1e-8, Noise added to the privacy loss
- **alpha** – float Gradient penalty weight for real samples.
- **rho** – float DAG loss factor
- **l1\_g** – float = 0 l1 regularization loss for the generator
- **l1\_W** – float = 1 l1 regularization factor for l1\_g
- **struct\_learning\_enabled** – bool Enable DAG learning outside DECAF.
- **struct\_learning\_n\_iter** – int Number of iterations for the DAG search.
- **struct\_learning\_search\_method** – str DAG search strategy: hillclimb, pc, tree\_search, mmhc, exhaustive, d-struct
- **struct\_learning\_score** – str DAG search scoring strategy: k2, bdeu, bic, bds
- **struct\_max\_indegree** – int Max parents in the DAG.
- **encoder\_max\_clusters** – int Number of clusters used for tabular encoding
- **device** – Any = DEVICE torch device used for training.
- **arguments** (# Core Plugin) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress\_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling\_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

### Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("decaf", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

```
class Config
 Bases: object
 arbitrary_types_allowed = True
 validate_assignment = True
```

**fit**(*X*: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any

Training method the synthetic data plugin.

#### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.data_loader import _
↳GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "=", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()
```

**Returns** self

**classmethod fqdn**() → str

The Fully-Qualified name of the plugin.

**generate**(count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any) →

*synthcity.plugins.core.data\_loader.DataLoader*

Synthetic data generation method.

## Parameters

- **count** – optional int. The number of samples to generate. If None, it generated `len(reference_dataset)` samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

### Valid Operations:

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

### Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "==", 0),
>>>]
>>>)
>>>
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**get\_dag**(X: pandas.core.frame.DataFrame, struct\_learning\_search\_method: Optional[str] = None, as\_index: bool = False) → Any

**static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**static name()** → str

The name of the plugin.

**plot**(plt: Any, X: [synthcity.plugins.core.data\\_loader.DataLoader](#), count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any) → Any

Plot the real-synthetic distributions.

**Parameters**

- **plt** – output
- **X** – [DataLoader](#). The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]

Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(trial: Any, \*args: Any, \*\*kwargs: Any) → Dict[str, Any]

**save()** → bytes

**save\_dict()** → dict

**save\_to\_file**(path: [pathlib.Path](#)) → bytes

**schema()** → [synthcity.plugins.core.schema.Schema](#)

The reference schema

**schema\_includes**(other: Union[[synthcity.plugins.core.data\\_loader.DataLoader](#), [pandas.core.frame.DataFrame](#)]) → bool

Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – [DataLoader](#). The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema()** → [synthcity.plugins.core.schema.Schema](#)

The internal schema

**static type()** → str

The type of the plugin.

**static version()** → str

API version

**plugin**

alias of [synthcity.plugins.privacy.plugin\\_decaf.DECAFPlugin](#)

### 10.1.3 Domain adaptation

**[synthcity.plugins.domain\\_adaptation.plugin\\_radialgan](#) module**

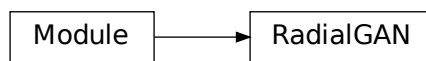
**Reference:** Yoon, Jinsung and Jordon, James and van der Schaar, Mihaela “RadialGAN: Leveraging multiple datasets to improve target-specific predictive models using Generative Adversarial Networks”

```

class RadialGAN(domains: List[int], n_features: int, n_units_latent: int, generator_n_layers_hidden: int = 2,
 generator_n_units_hidden: int = 250, generator_nonlin: str = 'leaky_relu',
 generator_nonlin_out: Optional[List[Tuple[str, int]]] = None, generator_n_iter: int = 500,
 generator_dropout: float = 0, generator_lr: float = 0.0002, generator_weight_decay: float =
 0.001, generator_opt_betas: tuple = (0.9, 0.999), discriminator_n_layers_hidden: int = 3,
 discriminator_n_units_hidden: int = 300, discriminator_nonlin: str = 'leaky_relu',
 discriminator_n_iter: int = 1, discriminator_dropout: float = 0.1, discriminator_lr: float =
 0.0002, discriminator_weight_decay: float = 0.001, discriminator_opt_betas: tuple = (0.9,
 0.999), batch_size: int = 64, n_iter_print: int = 10, random_state: int = 0, clipping_value: int
 = 0, lambda_gradient_penalty: float = 10, device: Any = device(type='cpu'),
 dataloader_sampler: Any = None)

```

Bases: `torch.nn.modules.module.Module`



RadialGAN implementation: Leveraging multiple datasets to improve target-specific predictive models using Generative Adversarial Networks.

#### Parameters

- **domains** – List[int] List of domains
- **n\_features** – int Number of features
- **n\_units\_latent** – int Number of hidden units
- **generator\_n\_layers\_hidden** – int Number of hidden layers in the generator
- **generator\_n\_units\_hidden** – int Number of hidden units in each layer of the Generator
- **generator\_nonlin** – string, default ‘elu’ Nonlinearity to use in the generator. Can be ‘elu’, ‘relu’, ‘selu’ or ‘leaky\_relu’.
- **generator\_n\_iter** – int Maximum number of iterations in the Generator.
- **generator\_dropout** – float Dropout value. If 0, the dropout is not used.
- **discriminator\_n\_layers\_hidden** – int Number of hidden layers in the discriminator
- **discriminator\_n\_units\_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator\_nonlin** – string, default ‘relu’ Nonlinearity to use in the discriminator. Can be ‘elu’, ‘relu’, ‘selu’ or ‘leaky\_relu’.
- **discriminator\_n\_iter** – int Maximum number of iterations in the discriminator.
- **discriminator\_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **lr** – float learning rate for optimizer.
- **weight\_decay** – float l2 (ridge) penalty for the weights.
- **batch\_size** – int Batch size

- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.
- **random\_state** – int random\_state used
- **clipping\_value** – int, default 0 Gradients clipping value. Zero disables the feature

**T\_destination**

alias of `TypeVar('T_destination', bound=Dict[str, Any])`

**add\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

**Parameters**

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

**apply**(*fn: Callable[[torch.nn.modules.module.Module], None]*) → `torch.nn.modules.module.T`

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** *fn* (*Module* → None) – function to be applied to each submodule

**Returns** self

**Return type** Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
```

**bfloat16**() → `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `bfloat16` datatype.

---

**Note:** This method modifies the module in-place.

---



**Returns** self

**Return type** Module

**buffers**(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**children**() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module

**cpu**() → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**dataloader**(*X: torch.Tensor, domains: torch.Tensor*) → torch.utils.data.dataloader.DataLoader

**double**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to double datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**dump\_patches:** bool = False

**eval()** → torch.nn.modules.module.T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** self

**Return type** Module

**extra\_repr()** → str

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**fit**(*X: numpy.ndarray, domains: numpy.ndarray*) →  
*synthcity.plugins.domain\_adaptation.plugin\_radialgan.RadialGAN*

**float()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to float datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**forward**(*count: int, domains: Optional[List[int]] = None*) → Tuple[torch.Tensor, List]

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**generate**(*count: int, domains: Optional[List[int]] = None*) → Tuple[numpy.ndarray, numpy.ndarray]

**get\_buffer**(*target: str*) → torch.Tensor

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

**Parameters** `target` – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by `target`

**Return type** torch.Tensor

**Raises `AttributeError`** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state()** → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter(target: str)** → torch.nn.parameter.Parameter

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

**Parameters target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by `target`

**Return type** torch.nn.Parameter

**Raises `AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule(target: str)** → torch.nn.modules.module.Module

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)
```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters target** – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by `target`

**Return type** torch.nn.Module

**Raises `AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**half()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**ipu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**load\_state\_dict**(*state\_dict: Mapping[str, Any], strict: bool = True*)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

**Parameters**

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `True`

**Returns**

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** `NamedTuple` with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as `None` and its corresponding key exists in `state_dict`, `load_state_dict()` will raise a `RuntimeError`.

---

**modules()** → `Iterator[torch.nn.modules.module.Module]`

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, 1)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str, torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children**() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str, Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo: Optional[Set[torch.nn.modules.module.Module]] = None, prefix: str = "", remove\_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module

- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** *(str, Module)* – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, `l` will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters** **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_buffer**(name: str, tensor: Optional[torch.Tensor], persistent: bool = True) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

#### Parameters

- **name** (str) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (Tensor or None) – buffer to be registered. If `None`, then operations that run on buffers, such as `cuda`, are ignored. If `None`, the buffer is **not** included in the module's `state_dict`.
- **persistent** (bool) – whether the buffer is part of this module's `state_dict`.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_forward\_pre\_hook**(hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned (unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → `torch.utils.hooks.RemovableHandle`

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`



**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None  
Alias for [add\\_module\(\)](#).

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None  
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If *None*, then operations that run on parameters, such as [cuda](#), are ignored. If *None*, the parameter is **not** included in the module's [state\\_dict](#).

**requires\_grad\_**(*requires\_grad: bool = True*) → torch.nn.modules.module.T  
Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** **requires\_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: *True*.

**Returns** *self*

**Return type** Module

**set\_extra\_state**(*state: Any*)  
This function is called from [load\\_state\\_dict\(\)](#) to handle any extra state found within the *state\_dict*. Implement this function and a corresponding [get\\_extra\\_state\(\)](#) for your module if you need to store extra state within its *state\_dict*.

**Parameters** **state** (*dict*) – Extra state from the *state\_dict*

**share\_memory**() → torch.nn.modules.module.T  
See `torch.Tensor.share_memory_()`

**state\_dict**(\*args, *destination=None, prefix="", keep\_vars=False*)  
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to *None* are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently `state_dict()` also accepts positional arguments for *destination*, *prefix* and *keep\_vars* in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument `destination` as it is not designed for end-users.

### Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in `state_dict`. Default: `''`.
- **keep\_vars** (*bool, optional*) – by default the `Tensor`s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** `dict`

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(*\*args, \*\*kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(*device=None, dtype=None, non\_blocking=False*)

**to**(*dtype, non\_blocking=False*)

**to**(*tensor, non\_blocking=False*)

**to**(*memory\_format=torch.channels\_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

**Note:** This method modifies the module in-place.

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

- **memory\_format** (torch.memory\_format) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** self

**Return type** Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

**to\_empty**(\*, device: Union[str, torch.device]) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** **device** (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** **mode** (*bool*) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training: bool**

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **dst\_type** (*type or string*) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

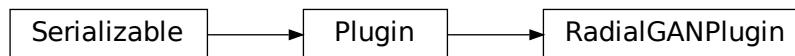
**zero\_grad**(*set\_to\_none: bool = False*) → None

Sets gradients of all model parameters to zero. See similar function under torch.optim.Optimizer for more context.

**Parameters** **set\_to\_none** (*bool*) – instead of setting to zero, set the grads to None. See torch.optim.Optimizer.zero\_grad() for details.

```
class RadialGANPlugin(n_iter: int = 2000, generator_n_layers_hidden: int = 2, generator_n_units_hidden: int = 500, generator_nonlin: str = 'relu', generator_dropout: float = 0.1, generator_opt_betas: tuple = (0.5, 0.999), discriminator_n_layers_hidden: int = 2, discriminator_n_units_hidden: int = 500, discriminator_nonlin: str = 'leaky_relu', discriminator_n_iter: int = 1, discriminator_dropout: float = 0.1, discriminator_opt_betas: tuple = (0.5, 0.999), lr: float = 0.001, weight_decay: float = 0.001, batch_size: int = 500, random_state: int = 0, clipping_value: int = 1, lambda_gradient_penalty: float = 10, encoder_max_clusters: int = 10, device: Any = device(type='cpu'), **kwargs: Any)
```

Bases: [synthcity.plugins.core.plugin.Plugin](#)



RadialGAN PyTorch implementation: Leveraging multiple datasets to improve target-specific predictive models using Generative Adversarial Networks.

#### Parameters

- **generator\_n\_layers\_hidden** – int Number of hidden layers in the generator
- **generator\_n\_units\_hidden** – int Number of hidden units in each layer of the Generator
- **generator\_nonlin** – string, default 'leaky\_relu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **n\_iter** – int Maximum number of iterations in the Generator.
- **generator\_dropout** – float Dropout value. If 0, the dropout is not used.
- **discriminator\_n\_layers\_hidden** – int Number of hidden layers in the discriminator
- **discriminator\_n\_units\_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator\_nonlin** – string, default 'leaky\_relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **discriminator\_n\_iter** – int Maximum number of iterations in the discriminator.
- **discriminator\_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **lr** – float learning rate for optimizer.
- **weight\_decay** – float l2 (ridge) penalty for the weights.
- **batch\_size** – int Batch size
- **random\_state** – int random seed to use
- **clipping\_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **encoder\_max\_clusters** – int The max number of clusters to create for continuous columns when encoding

### Example

```
>>> import numpy as np
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>> from synthcity.plugins.core.dataloader import GenericDataLoader
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>> X["domain"] = np.random.choice([0, 1], len(X)) # simulate domains
>>> dataloader = GenericDataLoader(X, domain_column="domain")
>>>
>>> plugin = Plugins().get("radialgan", n_iter = 100)
>>> plugin.fit(dataloader)
>>>
>>> plugin.generate(50)
```

### class Config

Bases: object

**arbitrary\_types\_allowed = True**

**validate\_assignment = True**

**fit**(X: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any

Training method the synthetic data plugin.

### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
```

(continues on next page)

(continued from previous page)

```

>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "==", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()

```

**Returns** self

**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any) → synthcity.plugins.core.data\_loader.DataLoader

Synthetic data generation method.

#### Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

#### Valid Operations:

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

#### Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "=", 0),
>>>]
>>>)
>>>
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**static name**() → str  
The name of the plugin.

**plot**(plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any) → Any  
Plot the real-synthetic distributions.

**Parameters**

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(trial: Any, \*args: Any, \*\*kwargs: Any) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(path: pathlib.Path) → bytes

**schema**() → synthcity.plugins.core.schema.Schema  
The reference schema

**schema\_includes**(other: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame]) → bool  
Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.



**training\_schema()** → *synthcity.plugins.core.schema.Schema*

The internal schema

**static type()** → str

The type of the plugin.

**static version()** → str

API version

```
class TabularRadialGAN(X: pandas.core.frame.DataFrame, domain_column: str, n_units_latent: int,
 generator_n_layers_hidden: int = 2, generator_n_units_hidden: int = 150,
 generator_nonlin: str = 'leaky_relu', generator_nonlin_out_discrete: str = 'softmax',
 generator_nonlin_out_continuous: str = 'none', generator_n_iter: int = 1000,
 generator_dropout: float = 0.01, generator_lr: float = 0.001, generator_weight_decay:
 float = 0.001, generator_opt_betas: tuple = (0.9, 0.999),
 discriminator_n_layers_hidden: int = 3, discriminator_n_units_hidden: int = 300,
 discriminator_nonlin: str = 'leaky_relu', discriminator_n_iter: int = 1,
 discriminator_dropout: float = 0.1, discriminator_lr: float = 0.001,
 discriminator_weight_decay: float = 0.001, discriminator_opt_betas: tuple = (0.9,
 0.999), batch_size: int = 64, n_iter_print: int = 100, random_state: int = 0,
 n_iter_min: int = 100, clipping_value: int = 0, lambda_gradient_penalty: float = 10,
 encoder_max_clusters: int = 20, device: Any = device(type='cpu'))
```

Bases: torch.nn.modules.module.Module

RadialGAN for tabular data.

This class combines RadialGAN and tabular encoder to form a generative model for tabular data.

#### Parameters

- **X** – pd.DataFrame Input dataset
- **domain\_column** – str The domain column
- **n\_units\_in** – int Number of features
- **generator\_n\_layers\_hidden** – int Number of hidden layers in the generator
- **generator\_n\_units\_hidden** – int Number of hidden units in each layer of the Generator
- **generator\_nonlin** – string, default 'elu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **generator\_n\_iter** – int Maximum number of iterations in the Generator.
- **generator\_dropout** – float Dropout value. If 0, the dropout is not used.
- **discriminator\_n\_layers\_hidden** – int Number of hidden layers in the discriminator
- **discriminator\_n\_units\_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator\_nonlin** – string, default 'relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **discriminator\_n\_iter** – int Maximum number of iterations in the discriminator.
- **discriminator\_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **lr** – float learning rate for optimizer..
- **weight\_decay** – float l2 (ridge) penalty for the weights.

- **batch\_size** – int Batch size
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.
- **random\_state** – int random\_state used
- **clipping\_value** – int, default 0 Gradients clipping value
- **lambda\_gradient\_penalty** – float Lambda weight for the gradient penalty
- **encoder\_max\_clusters** – int The max number of clusters to create for continuous columns when encoding

**T\_destination**

alias of `TypeVar('T_destination', bound=Dict[str, Any])`

**add\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

**Parameters**

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

**apply**(*fn: Callable[[torch.nn.modules.module.Module], None]*) → `torch.nn.modules.module.T`

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** *fn* (*Module* → None) – function to be applied to each submodule

**Returns** self

**Return type** Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**buffers**(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module

**cpu()** → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**decode**(*X: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

**double()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `double` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**dump\_patches:** bool = False

**encode**(*X: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

**eval()** → torch.nn.modules.module.T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** self

**Return type** Module

**extra\_repr()** → str

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**fit**(*X: pandas.core.frame.DataFrame*) → Any

**float()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `float` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**forward**(*count: int, domains: Optional[List[int]] = None*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**generate**(*count: int, domains: Optional[List[int]] = None*) → pandas.core.frame.DataFrame

**get\_buffer**(*target: str*) → torch.Tensor

Returns the buffer given by **target** if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify **target**.

**Parameters** **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by **target**

**Return type** torch.Tensor

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by **target** if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify **target**.

**Parameters** **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by **target**

**Return type** torch.nn.Parameter

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by **target** if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)
```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by `target`

**Return type** `torch.nn.Module`

**Raises** **`AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**`half()`**  $\rightarrow$  `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**`ipu(device: Optional[Union[int, torch.device]] = None)`**  $\rightarrow$  `torch.nn.modules.module.T`

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device (int, optional)` – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**`load_state_dict(state_dict: Mapping[str, Any], strict: bool = True)`**

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

**Parameters**

- **`state_dict (dict)`** – a dict containing parameters and persistent buffers.
- **`strict (bool, optional)`** – whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `True`

**Returns**

- **`missing_keys`** is a list of str containing the missing keys
- **`unexpected_keys`** is a list of str containing the unexpected keys

**Return type** `NamedTuple` with `missing_keys` and `unexpected_keys` fields

**Note:** If a parameter or buffer is registered as `None` and its corresponding key exists in `state_dict`, `load_state_dict()` will raise a `RuntimeError`.

**modules()**  $\rightarrow$  `Iterator[torch.nn.modules.module.Module]`

Returns an iterator over all modules in the network.

**Yields** `Module` – a module in the network

**Note:** Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix: str = "", recurse: bool = True*)  $\rightarrow$  `Iterator[Tuple[str, torch.Tensor]]`

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

**Parameters**

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if `True`, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str, torch.Tensor*) – `Tuple` containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children()**  $\rightarrow$  `Iterator[Tuple[str, torch.nn.modules.module.Module]]`

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str, Module*) – `Tuple` containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(memo: Optional[Set[torch.nn.modules.module.Module]] = None, prefix: str = "", remove\_duplicate: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (str, Module) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, l will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(prefix: str = "", recurse: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (str) – prefix to prepend to all parameter names.
- **recurse** (bool) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (str, Parameter) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(recurse: bool = True) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (bool) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.



**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_buffer**(name: str, tensor: Optional[torch.Tensor], persistent: bool = True) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm’s `running_mean` is not a parameter, but is part of the module’s state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module’s `state_dict`.

Buffers can be accessed as attributes using given names.

**Parameters**

- **name** (str) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (Tensor or None) – buffer to be registered. If `None`, then operations that run on buffers, such as `cuda`, are ignored. If `None`, the buffer is **not** included in the module’s `state_dict`.
- **persistent** (bool) – whether the buffer is part of this module’s `state_dict`.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`  
Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned (unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → `torch.utils.hooks.RemovableHandle`

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)  
Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The module argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None  
Alias for `add_module()`.

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None  
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad\_**(*requires\_grad: bool = True*) → `torch.nn.modules.module.T`  
Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** **requires\_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** `self`

**Return type** `Module`

**set\_extra\_state**(*state: Any*)  
This function is called from `load_state_dict()` to handle any extra state found within the `state_dict`. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its `state_dict`.

**Parameters** **state** (*dict*) – Extra state from the `state_dict`

**share\_memory\_**() → `torch.nn.modules.module.T`  
See `torch.Tensor.share_memory_()`

**state\_dict**(\*args, *destination=None, prefix="", keep\_vars=False*)  
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument `destination` as it is not designed for end-users.

### Parameters

- **destination** (*dict*, *optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str*, *optional*) – a prefix added to parameter and buffer names to compose the keys in `state_dict`. Default: `''`.
- **keep\_vars** (*bool*, *optional*) – by default the `Tensor`s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(\*args, \*\*kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(*device=None*, *dtype=None*, *non\_blocking=False*)

**to**(*dtype*, *non\_blocking=False*)

**to**(*tensor*, *non\_blocking=False*)

**to**(*memory\_format=torch.channels\_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module

- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** self

**Return type** Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

**to\_empty**(\*, *device*: `Union[str, torch.device]`) → `torch.nn.modules.module.T`  
 Moves the parameters and buffers to the specified device without copying storage.

**Parameters** `device` (`torch.device`) – The desired device of the parameters and buffers in this module.

**Returns** `self`

**Return type** `Module`

**train**(*mode: bool = True*) → `torch.nn.modules.module.T`

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

**Parameters** `mode` (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`).  
Default: `True`.

**Returns** `self`

**Return type** `Module`

**training:** `bool`

**type**(*dst\_type: Union[torch.dtype, str]*) → `torch.nn.modules.module.T`

Casts all parameters and buffers to `dst_type`.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `dst_type` (*type or string*) – the desired type

**Returns** `self`

**Return type** `Module`

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → `torch.nn.modules.module.T`

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**zero\_grad**(*set\_to\_none: bool = False*) → `None`

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** `set_to_none` (*bool*) – instead of setting to zero, set the grads to `None`. See `torch.optim.Optimizer.zero_grad()` for details.

**plugin**

alias of `synthcity.plugins.domain_adaptation.plugin_radialgan.RadialGANPlugin`

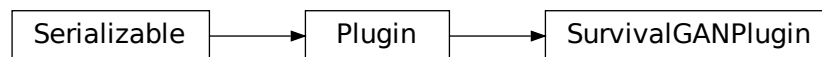
## 10.1.4 Static Survival Analysis

### synthcity.plugins.survival\_analysis.plugin\_survival\_gan module

Reference: “SurvivalGAN: Generating time-to-event Data for Survival Analysis”, B. Cebere\*, A. Norcliffe\*, F. Imrie, M. van der Schaar, AISTATS 2023

```
class SurvivalGANPlugin(uncensoring_model: str = 'survival_function_regression',
 dataloader_sampling_strategy: str = 'imbalanced_time_censoring', tte_strategy: str = 'survival_function', censoring_strategy: str = 'random', device: Any = device(type='cpu'), use_survival_conditional: bool = True, workspace: pathlib.Path = PosixPath('workspace'), random_state: int = 0, compress_dataset: bool = False, sampling_patience: int = 500, **kwargs: Any)
```

Bases: [synthcity.plugins.core.plugin.Plugin](#)



Survival Analysis Pipeline based on AdsGAN.

#### Parameters

- **uncensoring\_model** – str The time-to-event model: “survival\_function\_regression”.
- **dataloader\_sampling\_strategy** – str, default = imbalanced\_time\_censoring Train-ing sampling strategy: none, imbalanced\_censoring, imbalanced\_time\_censoring
- **tte\_strategy** – str The time-to-event generation strategy: survival\_function, uncensor-ing.
- **censoring\_strategy** – str For the generated data, how to censor subjects: “random” or “covariate\_dependent”
- **device** – torch device to use for training(cpu/cuda)
- **kwargs** – Any “adsgan” additional args, like n\_iter = 100 etc.
- **arguments** (# Core Plugin) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress\_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling\_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

### Example

```
>>> from lifelines.datasets import load_rossi
>>> from synthcity.plugins import Plugins
>>> from synthcity.plugins.core.dataloader import SurvivalAnalysisDataLoader
>>>
>>> X = load_rossi()
>>> data = SurvivalAnalysisDataLoader(
>>> X,
>>> target_column="arrest",
>>> time_to_event_column="week",
>>>)
>>>
>>> plugin = Plugins().get("survival_gan")
>>> plugin.fit(data)
>>>
>>> plugin.generate(count = 50)
```

### class Config

Bases: object

**arbitrary\_types\_allowed = True**

**validate\_assignment = True**

**fit**(X: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any

Training method the synthetic data plugin.

### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import
↳ GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
```

(continues on next page)



(continued from previous page)

```

>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "=", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()

```

**Returns** self**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any) → synthcity.plugins.core.data\_loader.DataLoader

Synthetic data generation method.

**Parameters**

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

**Valid Operations:**

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "=", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

**Usage example:**

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "=", 0),
>>>]
>>>)
>>>
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**static name**() → str  
The name of the plugin.

**plot**(plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any) → Any  
Plot the real-synthetic distributions.

**Parameters**

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(trial: Any, \*args: Any, \*\*kwargs: Any) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(path: pathlib.Path) → bytes

**schema**() → synthcity.plugins.core.schema.Schema  
The reference schema

**schema\_includes**(other: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame]) → bool  
Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema()** → *synthcity.plugins.core.schema.Schema*

The internal schema

**static type()** → str

The type of the plugin.

**static version()** → str

API version

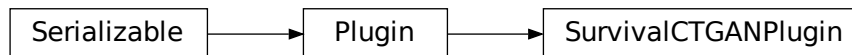
**plugin**

alias of *synthcity.plugins.survival\_analysis.plugin\_survival\_gan.SurvivalGANPlugin*

## synthcity.plugins.survival\_analysis.plugin\_survival\_ctgan module

```
class SurvivalCTGANPlugin(uncensoring_model: str = 'survival_function_regression',
 dataloader_sampling_strategy: str = 'imbalanced_time_censoring', tte_strategy:
 str = 'survival_function', censoring_strategy: str = 'random', device: Any =
 device(type='cpu'), workspace: pathlib.Path = PosixPath('workspace'),
 random_state: int = 0, compress_dataset: bool = False, sampling_patience: int =
 500, **kwargs: Any)
```

Bases: *synthcity.plugins.core.plugin.Plugin*



Survival Analysis Pipeline based on Conditional Tabular GANs.

### Parameters

- **uncensoring\_model** – str The time-to-event model: “survival\_function\_regression”.
- **dataloader\_sampling\_strategy** – str, default = imbalanced\_time\_censoring Training sampling strategy: none, imbalanced\_censoring, imbalanced\_time\_censoring
- **tte\_strategy** – str The time-to-event generation strategy: survival\_function, uncensoring.
- **censoring\_strategy** – str For the generated data, how to censor subjects: “random” or “covariate\_dependent”
- **device** – torch device to use for training(cpu/cuda)
- **kwargs** – Any “ctgan” additional args, like n\_iter = 100 etc.
- **arguments** (# Core Plugin) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress\_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling\_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

### Example

```
>>> from lifelines.datasets import load_rossi
>>> from synthcity.plugins import Plugins
>>> from synthcity.plugins.core.dataloader import SurvivalAnalysisDataLoader
>>>
>>> X = load_rossi()
>>> data = SurvivalAnalysisDataLoader(
>>> X,
>>> target_column="arrest",
>>> time_to_event_column="week",
>>>)
>>>
>>> plugin = Plugins().get("survival_ctgan")
>>> plugin.fit(data)
>>>
>>> plugin.generate(count = 50)
```

### class Config

Bases: object

**arbitrary\_types\_allowed = True**

**validate\_assignment = True**

**fit**(X: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any

Training method the synthetic data plugin.

### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import
↳ GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
```

(continues on next page)

(continued from previous page)

```

>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "=", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()

```

**Returns** self**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any) → synthcity.plugins.core.data\_loader.DataLoader

Synthetic data generation method.

**Parameters**

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

**Valid Operations:**

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "=", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

**Usage example:**

```

>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "=", 0),
>>>]
>>>)
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>> assert (syn_data["InterestingFeature"] == 0).all()

```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**static name**() → str  
The name of the plugin.

**plot**(plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any) → Any  
Plot the real-synthetic distributions.

**Parameters**

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(trial: Any, \*args: Any, \*\*kwargs: Any) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(path: pathlib.Path) → bytes

**schema**() → synthcity.plugins.core.schema.Schema  
The reference schema

**schema\_includes**(other: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame]) → bool  
Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema()** → *synthcity.plugins.core.schema.Schema*

The internal schema

**static type()** → str

The type of the plugin.

**static version()** → str

API version

**plugin**

alias of *synthcity.plugins.survival\_analysis.plugin\_survival\_ctgan.SurvivalCTGANPlugin*

## synthcity.plugins.survival\_analysis.plugin\_survae module

```
class SurVAEPlugin(uncensoring_model: str = 'survival_function_regression', dataloader_sampling_strategy: str = 'imbalanced_time_censoring', tte_strategy: str = 'survival_function', censoring_strategy: str = 'random', device: Any = device(type='cpu'), workspace: pathlib.Path = PosixPath('workspace'), random_state: int = 0, compress_dataset: bool = False, sampling_patience: int = 500, **kwargs: Any)
```

Bases: *synthcity.plugins.core.plugin.Plugin*



Survival Analysis Pipeline based on Variational AutoEncoders.

### Parameters

- **uncensoring\_model** –

**str** The time-to-event model: “survival\_function\_regression”.

**dataloader\_sampling\_strategy: str, default = imbalanced\_time\_censoring** Training sampling strategy: none, imbalanced\_censoring, imbalanced\_time\_censoring

**tte\_strategy: str**

The time-to-event generation strategy: survival\_function, uncensoring.

**censoring\_strategy: str** For the generated data, how to censor subjects: “random” or “covariate\_dependent”

- **device** – torch device to use for training(cpu/cuda) kwargs: Any

”tvae” additional args, like n\_iter = 100 etc.

# Core Plugin arguments workspace: Path.

Optional Path for caching intermediary results.

**compress\_dataset: bool. Default = False.** Drop redundant features before training the generator.

**sampling\_patience: int.** Max inference iterations to wait for the generated data to match the training schema.

### Example

```
>>> from lifelines.datasets import load_rossi
>>> from synthcity.plugins import Plugins
>>> from synthcity.plugins.core.dataloader import SurvivalAnalysisDataLoader
>>>
>>> X = load_rossi()
>>> data = SurvivalAnalysisDataLoader(
>>> X,
>>> target_column="arrest",
>>> time_to_event_column="week",
>>>)
>>>
>>> plugin = Plugins().get("survae")
>>> plugin.fit(data)
>>>
>>> plugin.generate(count = 50)
```

### class Config

Bases: object

**arbitrary\_types\_allowed = True**

**validate\_assignment = True**

**fit**(X: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any

Training method the synthetic data plugin.

#### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import _
↳ GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
```

(continues on next page)



(continued from previous page)

```

>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "=", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()

```

Returns self

**classmethod** `fqdn()` → str

The Fully-Qualified name of the plugin.

**generate**(*count*: Optional[int] = None, *constraints*: Optional[synthcity.plugins.core.constraints.Constraints] = None, *random\_state*: Optional[int] = None, *\*\*kwargs*: Any) → *synthcity.plugins.core.data\_loader.DataLoader*

Synthetic data generation method.

#### Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

#### Valid Operations:

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "=", "eq": equal with <value>

- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "==", 0),
>>>]
>>>)
>>>
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**static name**() → str  
The name of the plugin.

**plot**(plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any) → Any  
Plot the real-synthetic distributions.

**Parameters**

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(trial: Any, \*args: Any, \*\*kwargs: Any) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(path: pathlib.Path) → bytes

**schema**() → synthcity.plugins.core.schema.Schema  
The reference schema

**schema\_includes**(*other*: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame]) → bool

Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema**() → synthcity.plugins.core.schema.Schema

The internal schema

**static type**() → str

The type of the plugin.

**static version**() → str

API version

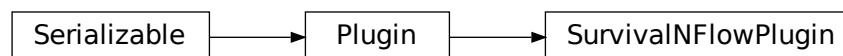
**plugin**

alias of `synthcity.plugins.survival_analysis.plugin_survae.SurVAEPlugin`

## synthcity.plugins.survival\_analysis.plugin\_survival\_nflow module

```
class SurvivalNFlowPlugin(uncensoring_model: str = 'survival_function_regression',
 dataloader_sampling_strategy: str = 'imbalanced_time_censoring', tte_strategy:
 str = 'survival_function', censoring_strategy: str = 'random', device: Any =
 device(type='cpu'), workspace: pathlib.Path = PosixPath('workspace'),
 random_state: int = 0, compress_dataset: bool = False, sampling_patience: int =
 500, **kwargs: Any)
```

Bases: `synthcity.plugins.core.plugin.Plugin`



Survival Analysis Pipeline based on Normalizing flows.

**Parameters** **uncensoring\_model** –

**str** The time-to-event model: “survival\_function\_regression”.

**dataloader\_sampling\_strategy: str, default = imbalanced\_time\_censoring** Training sampling strategy: none, imbalanced\_censoring, imbalanced\_time\_censoring

**tte\_strategy: str**

The time-to-event generation strategy: survival\_function, uncensoring.

**censoring\_strategy: str** For the generated data, how to censor subjects: “random” or “covariate\_dependent”

**kwargs: Any** ”nflow” additional args, like n\_iter = 100 etc.

**device:** torch device to use for training(cpu/cuda)

# Core Plugin arguments workspace: Path.

Optional Path for caching intermediary results.

**compress\_dataset:** bool. **Default = False.** Drop redundant features before training the generator.

**sampling\_patience:** int. Max inference iterations to wait for the generated data to match the training schema.

### Example

```
>>> from lifelines.datasets import load_rossi
>>> from synthcity.plugins import Plugins
>>> from synthcity.plugins.core.data_loader import SurvivalAnalysisDataLoader
>>>
>>> X = load_rossi()
>>> data = SurvivalAnalysisDataLoader(
>>> X,
>>> target_column="arrest",
>>> time_to_event_column="week",
>>>)
>>>
>>> plugin = Plugins().get("survival_ctgan")
>>> plugin.fit(data)
>>>
>>> plugin.generate(count = 50)
```

### class Config

Bases: object

**arbitrary\_types\_allowed = True**

**validate\_assignment = True**

**fit**(X: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any

Training method the synthetic data plugin.

### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.data_loader import _
↳ GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
```

(continues on next page)

(continued from previous page)

```

>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "=", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()

```

**Returns** self**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(*count*: Optional[int] = None, *constraints*: Optional[synthcity.plugins.core.constraints.Constraints] = None, *random\_state*: Optional[int] = None, *\*\*kwargs*: Any) →  
 synthcity.plugins.core.data\_loader.DataLoader

Synthetic data generation method.

**Parameters**

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

**Valid Operations:**

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>

- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

**Usage example:**

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "==", 0),
>>>]
>>>)
>>>
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**static name**() → str  
The name of the plugin.

**plot**(plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any) → Any  
Plot the real-synthetic distributions.

**Parameters**

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(trial: Any, \*args: Any, \*\*kwargs: Any) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(path: pathlib.Path) → bytes

**schema()** → *synthcity.plugins.core.schema.Schema*

The reference schema

**schema\_includes**(*other: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame]*) → bool

Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema()** → *synthcity.plugins.core.schema.Schema*

The internal schema

**static type()** → str

The type of the plugin.

**static version()** → str

API version

**plugin**

alias of *synthcity.plugins.survival\_analysis.plugin\_survival\_nflow.SurvivalNFlowPlugin*

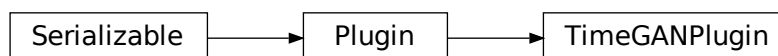
## 10.1.5 Time-series & Time-Series Survival Analysis

### synthcity.plugins.time\_series.plugin\_timegan module

Reference: “Time-series Generative Adversarial Networks”, Jinsung Yoon, Daniel Jarrett, Mihaela van der Schaar

**class TimeGANPlugin**(*n\_iter: int = 1000, generator\_n\_layers\_hidden: int = 2, generator\_n\_units\_hidden: int = 150, generator\_nonlin: str = 'leaky\_relu', generator\_nonlin\_out\_discrete: str = 'softmax', generator\_nonlin\_out\_continuous: str = 'tanh', generator\_batch\_norm: bool = False, generator\_dropout: float = 0.01, generator\_loss: Optional[Callable] = None, generator\_lr: float = 0.001, generator\_weight\_decay: float = 0.001, generator\_residual: bool = True, discriminator\_n\_layers\_hidden: int = 3, discriminator\_n\_units\_hidden: int = 300, discriminator\_nonlin: str = 'leaky\_relu', discriminator\_n\_iter: int = 1, discriminator\_batch\_norm: bool = False, discriminator\_dropout: float = 0.1, discriminator\_loss: Optional[Callable] = None, discriminator\_lr: float = 0.001, discriminator\_weight\_decay: float = 0.001, batch\_size: int = 64, n\_iter\_print: int = 10, clipping\_value: int = 0, encoder\_max\_clusters: int = 20, encoder: Optional[Any] = None, device: Any = device(type='cpu'), mode: str = 'RNN', gamma\_penalty: float = 1, moments\_penalty: float = 100, embedding\_penalty: float = 10, use\_horizon\_condition: bool = True, dataloader\_sampling\_strategy: str = 'imbalanced\_time\_censoring', random\_state: int = 0, workspace: pathlib.Path = PosixPath('workspace'), compress\_dataset: bool = False, sampling\_patience: int = 500, \*\*kwargs: Any)*)

Bases: *synthcity.plugins.core.plugin.Plugin*



Synthetic time series generation using TimeGAN.

### Parameters

- **n\_iter** – int Maximum number of iterations in the Generator.
- **n\_units\_in** – int Number of features
- **generator\_n\_layers\_hidden** – int Number of hidden layers in the generator
- **generator\_n\_units\_hidden** – int Number of hidden units in each layer of the Generator
- **generator\_nonlin** – string, default 'elu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **generator\_batch\_norm** – bool Enable/disable batch norm for the generator
- **generator\_dropout** – float Dropout value. If 0, the dropout is not used.
- **generator\_residual** – bool Use residuals for the generator
- **discriminator\_n\_layers\_hidden** – int Number of hidden layers in the discriminator
- **discriminator\_n\_units\_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator\_nonlin** – string, default 'relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **discriminator\_n\_iter** – int Maximum number of iterations in the discriminator.
- **discriminator\_batch\_norm** – bool Enable/disable batch norm for the discriminator
- **discriminator\_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **lr** – float learning rate for optimizer.
- **weight\_decay** – float l2 (ridge) penalty for the weights.
- **batch\_size** – int Batch size
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.
- **random\_state** – int random\_state used
- **clipping\_value** – int, default 0 Gradients clipping value
- **gamma\_penalty** – Latent representation penalty
- **moments\_penalty** – float = 100 Moments(var and mean) penalty
- **embedding\_penalty** – float = 10 Embedding representation penalty
- **mode** – str = "RNN" Core neural net architecture. Available models:
  - "LSTM"
  - "GRU"
  - "RNN"
  - "Transformer"
  - "MLSTM\_FCN"
  - "TCN"
  - "InceptionTime"



- "InceptionTimePlus"
- "XceptionTime"
- "ResCNN"
- "OmniScaleCNN"
- "XCM"
- **device** – The device used by PyTorch. cpu/cuda
- **use\_horizon\_condition** – bool. Default = True Whether to condition the covariate generation on the observation times or not.
- **encoder\_max\_clusters** – int The max number of clusters to create for continuous columns when encoding
- **encoder** – Pre-trained tabular encoder. If None, a new encoder is trained.
- **arguments** (# Core Plugin) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress\_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling\_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

### Example

```
>>> from synthcity.plugins import Plugins
>>> from synthcity.utils.datasets.time_series.google_stocks import _
↳ GoogleStocksDataLoader
>>> from synthcity.plugins.core.data_loader import TimeSeriesDataLoader
>>> static, temporal, horizons, outcome = GoogleStocksDataLoader().load()
>>> loader = TimeSeriesDataLoader(
>>> temporal_data=temporal,
>>> observation_times=horizons,
>>> static_data=static,
>>> outcome=outcome,
>>>)
>>>
>>> plugin = Plugins().get("timegan", n_iter = 50)
>>> plugin.fit(loader)
>>>
>>> plugin.generate(count = 10)
```

### class Config

Bases: object

**arbitrary\_types\_allowed** = True

**validate\_assignment** = True

**fit**(X: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame], \*args: Any,

\*\*kwargs: Any) → Any

Training method the synthetic data plugin.

**Parameters**

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.data_loader import _
↳ GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "=", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()
```

**Returns** self

**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any) → synthcity.plugins.core.data\_loader.DataLoader

Synthetic data generation method.

#### Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation

Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.

- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

#### Valid Operations:

- "<", "lt": less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt": greater than <value>
- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

#### Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "==", 0),
>>>]
>>>)
>>>
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**static name**() → str  
The name of the plugin.

**plot**(plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any) → Any  
Plot the real-synthetic distributions.

#### Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]

Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(trial: Any, \*args: Any, \*\*kwargs: Any) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(path: pathlib.Path) → bytes

**schema**() → [synthcity.plugins.core.schema.Schema](#)  
The reference schema

**schema\_includes**(other: Union[[synthcity.plugins.core.data\\_loader.DataLoader](#),  
[pandas.core.frame.DataFrame](#)]) → bool  
Helper method to test if the reference schema includes a Dataset

**Parameters** other – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema**() → [synthcity.plugins.core.schema.Schema](#)  
The internal schema

**static type**() → str  
The type of the plugin.

**static version**() → str  
API version

**plugin**

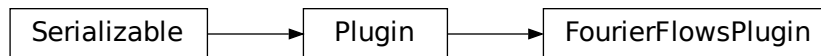
alias of [synthcity.plugins.time\\_series.plugin\\_timegan.TimeGANPlugin](#)

## **synthcity.plugins.time\_series.plugin\_fflows module**

Reference: “Generative Time-series Modeling with Fourier Flows”, Ahmed Alaa, Alex Chan, and Mihaela van der Schaar.

**class FourierFlowsPlugin**(n\_iter: int = 500, batch\_size: int = 128, lr: float = 0.001, n\_iter\_print: int = 100,  
n\_units\_hidden: int = 100, n\_flows: int = 10, FFT: bool = True, flip: bool = True,  
normalize: bool = False, static\_model: str = 'ctgan', device: Any =  
device(type='cpu'), encoder\_max\_clusters: int = 10, random\_state: int = 0,  
workspace: pathlib.Path = PosixPath('workspace'), compress\_dataset: bool = False,  
sampling\_patience: int = 500, \*\*kwargs: Any)

Bases: [synthcity.plugins.core.plugin.Plugin](#)



Synthetic time series generation using FourierFlows.

### Parameters

- **n\_iter** – int Number of training iterations
- **batch\_size** – int Batch size
- **lr** – float Learning rate
- **n\_iter\_print** – int Number of iterations to print the validation loss
- **n\_units\_hidden** – int Number of hidden nodes
- **n\_flows** – int Number of flows to use(default = 10)
- **FFT** – bool Use Fourier transform(default = True)
- **flip** – bool Flip the data in the SpectralFilter
- **normalize** – bool Scale the data(default = False)
- **static\_model** – str = “ctgan”, The model to use for generating the static data.
- **device** – Any = DEVICE torch device to use for training(cpu/cuda)
- **encoder\_max\_clusters** – int = 10 Number of clusters used for tabular encoding
- **arguments** (# Core Plugin) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress\_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling\_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

### Example

```

>>> from synthcity.plugins import Plugins
>>> from synthcity.utils.datasets.time_series.google_stocks import _
↳ GoogleStocksDataloader
>>> from synthcity.plugins.core.dataloader import TimeSeriesDataLoader
>>> static, temporal, horizons, outcome = GoogleStocksDataloader().load()
>>> loader = TimeSeriesDataLoader(
>>> temporal_data=temporal,
>>> observation_times=horizons,
>>> static_data=static,
>>> outcome=outcome,
>>>)
>>>

```

(continues on next page)

(continued from previous page)

```
>>> plugin = Plugins().get("ffflows", n_iter = 50)
>>> plugin.fit(loader)
>>>
>>> plugin.generate(count = 10)
```

**class Config**

Bases: object

**arbitrary\_types\_allowed = True****validate\_assignment = True**

**fit**(X: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any

Training method the synthetic data plugin.

**Parameters**

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.data_loader import _
↳GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "==", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
```

(continues on next page)

(continued from previous page)

```
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()
```

**Returns** self

**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(*count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any*) → *synthcity.plugins.core.data\_loader.DataLoader*

Synthetic data generation method.

#### Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

#### Valid Operations:

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

#### Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "==", 0),
>>>]
>>>)
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
```

(continues on next page)

(continued from previous page)

```
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**static name**() → str  
The name of the plugin.

**plot**(plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any) → Any  
Plot the real-synthetic distributions.

#### Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(trial: Any, \*args: Any, \*\*kwargs: Any) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(path: pathlib.Path) → bytes

**schema**() → synthcity.plugins.core.schema.Schema  
The reference schema

**schema\_includes**(other: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame]) → bool  
Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema**() → synthcity.plugins.core.schema.Schema  
The internal schema

**static type**() → str  
The type of the plugin.

**static version**() → str  
API version

#### plugin

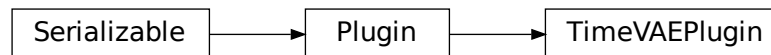
alias of `synthcity.plugins.time_series.plugin_fflows.FourierFlowsPlugin`



## synthcity.plugins.time\_series.plugin\_timevae module

```
class TimeVAEPlugin(n_iter: int = 1000, decoder_n_layers_hidden: int = 2, decoder_n_units_hidden: int = 150,
 decoder_nonlin: str = 'leaky_relu', decoder_nonlin_out_discrete: str = 'softmax',
 decoder_nonlin_out_continuous: str = 'tanh', decoder_batch_norm: bool = False,
 decoder_dropout: float = 0.01, decoder_residual: bool = True, encoder_n_layers_hidden:
 int = 3, encoder_n_units_hidden: int = 300, encoder_nonlin: str = 'leaky_relu',
 encoder_batch_norm: bool = False, encoder_dropout: float = 0.1, lr: float = 0.001,
 weight_decay: float = 0.001, batch_size: int = 64, n_iter_print: int = 10, clipping_value:
 int = 0, encoder_max_clusters: int = 20, encoder: Optional[Any] = None, device: Any =
 device(type='cpu'), mode: str = 'LSTM', gamma_penalty: float = 1, moments_penalty:
 float = 100, embedding_penalty: float = 10, random_state: int = 0, workspace:
 pathlib.Path = PosixPath('workspace'), compress_dataset: bool = False,
 sampling_patience: int = 500, **kwargs: Any)
```

Bases: [synthcity.plugins.core.plugin.Plugin](#)



Synthetic time series generation using a Variational AutoEncoder.

### Parameters

- **n\_iter** – int Maximum number of iterations in the decoder.
- **n\_units\_in** – int Number of features
- **decoder\_n\_layers\_hidden** – int Number of hidden layers in the decoder
- **decoder\_n\_units\_hidden** – int Number of hidden units in each layer of the decoder
- **decoder\_nonlin** – string, default 'elu' Nonlinearity to use in the decoder. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **decoder\_batch\_norm** – bool Enable/disable batch norm for the decoder
- **decoder\_dropout** – float Dropout value. If 0, the dropout is not used.
- **decoder\_residual** – bool Use residuals for the decoder
- **encoder\_n\_layers\_hidden** – int Number of hidden layers in the encoder
- **encoder\_n\_units\_hidden** – int Number of hidden units in each layer of the encoder
- **encoder\_nonlin** – string, default 'relu' Nonlinearity to use in the encoder. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **encoder\_n\_iter** – int Maximum number of iterations in the encoder.
- **encoder\_batch\_norm** – bool Enable/disable batch norm for the encoder
- **encoder\_dropout** – float Dropout value for the encoder. If 0, the dropout is not used.
- **lr** – float learning rate for optimizer.
- **weight\_decay** – float l2 (ridge) penalty for the weights.

- **batch\_size** – int Batch size
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.
- **random\_state** – int random\_state used
- **clipping\_value** – int, default 0 Gradients clipping value
- **mode** – str = “RNN” Core neural net architecture. Available models:
  - “LSTM”
  - “GRU”
  - “RNN”
  - “Transformer”
  - “MLSTM\_FCN”
  - “TCN”
  - “InceptionTime”
  - “InceptionTimePlus”
  - “XceptionTime”
  - “ResCNN”
  - “OmniScaleCNN”
  - “XCM”
- **device** – The device used by PyTorch. cpu/cuda
- **use\_horizon\_condition** – bool. Default = True Whether to condition the covariate generation on the observation times or not.
- **encoder\_max\_clusters** – int The max number of clusters to create for continuous columns when encoding
- **encoder** – Pre-trained tabular encoder. If None, a new encoder is trained.
- **arguments** (# Core Plugin) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress\_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling\_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

### Example

```
>>> from synthcity.plugins import Plugins
>>> from synthcity.utils.datasets.time_series.google_stocks import _
↳ GoogleStocksDataLoader
>>> from synthcity.plugins.core.data_loader import TimeSeriesDataLoader
>>>
>>> plugin = Plugins().get("timevae")
>>> static, temporal, outcome = GoogleStocksDataLoader(as_numpy=True).load()
```

(continues on next page)

(continued from previous page)

```
>>> loader = TimeSeriesDataLoader(
>>> temporal_data=temporal_data,
>>> static_data=static_data,
>>> outcome=outcome,
>>>)
>>> plugin.fit(loader)
>>> plugin.generate()
```

**class Config**

Bases: object

**arbitrary\_types\_allowed = True****validate\_assignment = True**

**fit**(X: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any

Training method the synthetic data plugin.

**Parameters**

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.data_loader import _
↳GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "=", 1),
>>>]
>>>)
```

(continues on next page)

(continued from previous page)

```

>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()

```

**Returns** self

**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(*count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any*) → *synthcity.plugins.core.data\_loader.DataLoader*

Synthetic data generation method.

#### Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

#### Valid Operations:

- "<", "lt": less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt": greater than <value>
- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

#### Usage example:

```

>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "==", 0),
>>>]
>>>)
>>> syn_data = syn_model.generate(

```

(continues on next page)

(continued from previous page)

```

 count=count,
 constraints=constraints
).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()

```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(*\*\*kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]  
Returns the hyperparameter space for the derived plugin.

**static load**(*buff: bytes*) → Any

**static load\_dict**(*representation: dict*) → Any

**static name**() → str  
The name of the plugin.

**plot**(*plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any*) → Any  
Plot the real-synthetic distributions.

#### Parameters

- **plt** – output
- **X** – *DataLoader*. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(*\*args: Any, \*\*kwargs: Any*) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(*trial: Any, \*args: Any, \*\*kwargs: Any*) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(*path: pathlib.Path*) → bytes

**schema**() → *synthcity.plugins.core.schema.Schema*  
The reference schema

**schema\_includes**(*other: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame]*) → bool  
Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – *DataLoader*. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema**() → *synthcity.plugins.core.schema.Schema*  
The internal schema

**static type**() → str  
The type of the plugin.

**static version**() → str  
API version

**plugin**

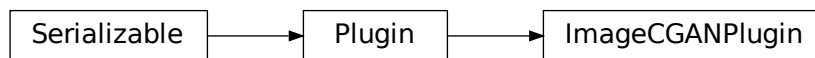
alias of `synthcity.plugins.time_series.plugin_timevae.TimeVAEPlugin`

## 10.1.6 Images

### `synthcity.plugins.images.plugin_image_cgan` module

```
class ImageCGANPlugin(n_units_latent: int = 100, n_iter: int = 1000, generator_nonlin: str = 'relu',
 generator_dropout: float = 0.1, generator_n_residual_units: int = 2,
 discriminator_nonlin: str = 'leaky_relu', discriminator_n_iter: int = 5,
 discriminator_dropout: float = 0.1, discriminator_n_residual_units: int = 2, lr: float =
0.0002, weight_decay: float = 0.001, opt_betas: tuple = (0.5, 0.999), batch_size: int =
200, random_state: int = 0, clipping_value: int = 1, lambda_gradient_penalty: float =
10, device: Any = device(type='cpu'), patience: int = 5, patience_metric:
Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None, n_iter_print:
int = 50, n_iter_min: int = 100, plot_progress: int = False, early_stopping: bool =
True, workspace: pathlib.Path = PosixPath('workspace'), sampling_patience: int = 500,
**kwargs: Any)
```

Bases: `synthcity.plugins.core.plugin.Plugin`



#### Image (Conditional) GAN

##### Parameters

- **n\_units\_latent** – int The noise units size used by the generator.
- **n\_iter** – int Maximum number of iterations in the Generator.
- **generator\_nonlin** – string, default 'leaky\_relu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **generator\_dropout** – float Dropout value. If 0, the dropout is not used.
- **generator\_n\_residual\_units** – int The number of convolutions in residual units for the generator, 0 means no residual units
- **discriminator\_nonlin** – string, default 'leaky\_relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **discriminator\_n\_iter** – int Maximum number of iterations in the discriminator.
- **discriminator\_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **discriminator\_n\_residual\_units** – int The number of convolutions in residual units for the discriminator, 0 means no residual units
- **parameters** (# training) –
- **lr** – float learning rate for optimizer..

- **weight\_decay** – float l2 (ridge) penalty for the weights.
- **batch\_size** – int Batch size
- **random\_state** – int random seed to use
- **clipping\_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **device** – torch device Device: cpu or cuda
- **plot\_progress** – bool Plot some synthetic samples every *n\_iter\_print*
- **stopping** (# *early*) –
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.
- **n\_iter\_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before training early stopping is trigged.
- **patience\_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for training early stopping.
- **early\_stopping** – bool Evaluate the quality of the synthetic data using *patience\_metric*, and stop after *patience* iteration with no improvement.
- **arguments** (# *Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.

### Example

```
>>> from torchvision import datasets
>>> from synthcity.plugins import Plugins
>>> from synthcity.plugins.core.dataloader import ImageDataLoader
>>>
>>> model = Plugins().get("image_cgan", n_iter = 10)
>>>
>>> dataset = datasets.MNIST(".", download=True)
>>> dataloader = ImageDataLoader(dataset).sample(100)
>>>
>>> model.fit(dataloader)
>>>
>>> X_gen = model.generate(50)
>>> assert len(X_gen) == 50
```

```
class Config
```

```
 Bases: object
```

```
 arbitrary_types_allowed = True
```

```
 validate_assignment = True
```

```
fit(X: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame], *args: Any,
 **kwargs: Any) → Any
```

```
 Training method the synthetic data plugin.
```

```
 Parameters
```

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.data_loader import _
GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "=", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()
```

**Returns** self

**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any) → synthcity.plugins.core.data\_loader.DataLoader

Synthetic data generation method.

#### Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation



Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.

- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

#### Valid Operations:

- "<", "lt": less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt": greater than <value>
- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

#### Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "==", 0),
>>>]
>>>)
>>>
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**static name**() → str  
The name of the plugin.

**plot**(plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any) → Any  
Plot the real-synthetic distributions.

#### Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod** **sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]

Sample value from the hyperparameter space for the current plugin.

**classmethod** **sample\_hyperparameters\_optuna**(trial: Any, \*args: Any, \*\*kwargs: Any) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(path: pathlib.Path) → bytes

**schema**() → *synthcity.plugins.core.schema.Schema*

The reference schema

**schema\_includes**(other: Union[*synthcity.plugins.core.dataloader.DataLoader*, *pandas.core.frame.DataFrame*]) → bool

Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema**() → *synthcity.plugins.core.schema.Schema*

The internal schema

**static type**() → str

The type of the plugin.

**static version**() → str

API version

**plugin**

alias of *synthcity.plugins.images.plugin\_image\_cgan.ImageCGANPlugin*

## **synthcity.plugins.images.plugin\_image\_adsgan module**

```
class ImageAdsGANPlugin(n_units_latent: int = 100, n_iter: int = 1000, generator_nonlin: str = 'relu',
 generator_dropout: float = 0.1, generator_n_residual_units: int = 2,
 discriminator_nonlin: str = 'leaky_relu', discriminator_n_iter: int = 5,
 discriminator_dropout: float = 0.1, discriminator_n_residual_units: int = 2, lr: float
 = 0.0002, weight_decay: float = 0.001, opt_betas: tuple = (0.5, 0.999), batch_size:
 int = 200, random_state: int = 0, clipping_value: int = 1, lambda_gradient_penalty:
 float = 10, lambda_identifiability_penalty: float = 0.1, device: Any =
 device(type='cpu'), patience: int = 5, patience_metric:
 Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None,
 n_iter_print: int = 50, n_iter_min: int = 100, plot_progress: int = False,
 early_stopping: bool = True, workspace: pathlib.Path = PosixPath('workspace'),
 sampling_patience: int = 500, **kwargs: Any)
```

Bases: *synthcity.plugins.core.plugin.Plugin*

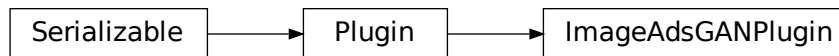


Image AdsGAN - Anonymization through Data Synthesis using Generative Adversarial Networks.

### Parameters

- **n\_units\_latent** – int The noise units size used by the generator.
- **n\_iter** – int Maximum number of iterations in the Generator.
- **generator\_nonlin** – string, default 'leaky\_relu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **generator\_dropout** – float Dropout value. If 0, the dropout is not used.
- **generator\_n\_residual\_units** – int The number of convolutions in residual units for the generator, 0 means no residual units
- **discriminator\_nonlin** – string, default 'leaky\_relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.
- **discriminator\_n\_iter** – int Maximum number of iterations in the discriminator.
- **discriminator\_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **discriminator\_n\_residual\_units** – int The number of convolutions in residual units for the discriminator, 0 means no residual units
- **parameters** (# *training*) –
- **lr** – float learning rate for optimizer
- **weight\_decay** – float l2 (ridge) penalty for the weights.
- **batch\_size** – int Batch size
- **random\_state** – int random seed to use
- **clipping\_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **lambda\_gradient\_penalty** – float = 10 Weight for the gradient penalty
- **lambda\_identifiability\_penalty** – float = 0.1 Weight for the identifiability penalty
- **device** – torch device Device: cpu or cuda
- **plot\_progress** – bool Plot some synthetic samples every *n\_iter\_print*
- **stopping** (# *early*) –
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.
- **n\_iter\_min** – int Minimum number of iterations to go through before starting early stopping
- **early\_stopping** – bool Evaluate the quality of the synthetic data using *patience\_metric*, and stop after *patience* iteration with no improvement.

- **patience** – int Max number of iterations without any improvement before training early stopping is triggered.
- **patience\_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for training early stopping.
- **arguments** (# Core Plugin) –
- **workspace** – Path. Optional Path for caching intermediary results.

### Example

```
>>> from torchvision import datasets
>>> from synthcity.plugins import Plugins
>>> from synthcity.plugins.core.data_loader import ImageDataLoader
>>>
>>> model = Plugins().get("image_adsgan", n_iter = 10)
>>>
>>> dataset = datasets.MNIST(".", download=True)
>>> dataloader = ImageDataLoader(dataset).sample(100)
>>>
>>> model.fit(dataloader)
>>>
>>> X_gen = model.generate(50)
>>> assert len(X_gen) == 50
```

### class Config

Bases: object

**arbitrary\_types\_allowed** = True

**validate\_assignment** = True

**fit**(X: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any

Training method the synthetic data plugin.

### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.data_loader import _
↳ GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
```

(continues on next page)

(continued from previous page)

```

>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "=", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()

```

**Returns** self

**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any) → synthcity.plugins.core.data\_loader.DataLoader

Synthetic data generation method.

#### Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

#### Valid Operations:

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>

- "==" , "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "==", 0),
>>>]
>>>)
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**static name**() → str  
The name of the plugin.

**plot**(plt: Any, X: synthcity.plugins.core.dataloader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any) → Any  
Plot the real-synthetic distributions.

**Parameters**

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(trial: Any, \*args: Any, \*\*kwargs: Any) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(path: pathlib.Path) → bytes

**schema**() → synthcity.plugins.core.schema.Schema  
The reference schema

**schema\_includes**(*other*: Union[synthcity.plugins.core.dataloader.DataLoader,  
pandas.core.frame.DataFrame]) → bool

Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema**() → synthcity.plugins.core.schema.Schema

The internal schema

**static type**() → str

The type of the plugin.

**static version**() → str

API version

**plugin**

alias of *synthcity.plugins.images.plugin\_image\_adsgan.ImageAdsGANPlugin*





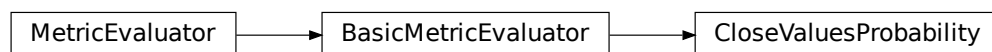
## 11.1 Metrics and benchmarks

### 11.1.1 synthcity.metrics.eval\_sanity module

```
class BasicMetricEvaluator(**kwargs: Any)
 Bases: synthcity.metrics.core.metric.MetricEvaluator
```



```
static direction() → str
evaluate(X_gt: synthcity.plugins.core.data_loader.DataLoader, X_syn:
 synthcity.plugins.core.data_loader.DataLoader) → float
evaluate_default(X_gt: synthcity.plugins.core.data_loader.DataLoader, X_syn:
 synthcity.plugins.core.data_loader.DataLoader) → float
classmethod fqdn() → str
static name() → str
reduction() → Callable
static type() → str
use_cache(path: pathlib.Path) → bool
class CloseValuesProbability(**kwargs: Any)
 Bases: synthcity.metrics.eval_sanity.BasicMetricEvaluator
```



Compute the probability of close values between the real and synthetic data.

**Score:** 0 means there is no chance to have synthetic rows similar to the real. 1 means that all the synthetic rows are similar to some real rows.

**static direction()** → str

**evaluate**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*:  
synthcity.plugins.core.dataloader.DataLoader) → Dict

**evaluate\_default**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*:  
synthcity.plugins.core.dataloader.DataLoader) → float

**classmethod fqdn()** → str

**static name()** → str

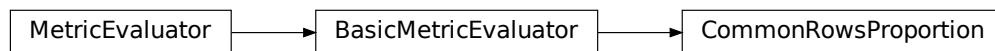
**reduction()** → Callable

**static type()** → str

**use\_cache**(*path*: pathlib.Path) → bool

**class CommonRowsProportion**(\*\*kwargs: Any)

Bases: [synthcity.metrics.eval\\_sanity.BasicMetricEvaluator](#)



Returns the proportion of rows in the real dataset leaked in the synthetic dataset.

**Score:** 0: there are no common rows between the real and synthetic datasets. 1: all the rows in the real dataset are leaked in the synthetic dataset.

**static direction()** → str

**evaluate**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*:  
synthcity.plugins.core.dataloader.DataLoader) → Dict

**evaluate\_default**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*:  
synthcity.plugins.core.dataloader.DataLoader) → float

**classmethod fqdn()** → str

**static name()** → str

**reduction()** → Callable

**static type()** → str

**use\_cache**(*path*: pathlib.Path) → bool

**class DataMismatchScore**(\*\*kwargs: Any)

Bases: [synthcity.metrics.eval\\_sanity.BasicMetricEvaluator](#)



Basic sanity score. Compares the data types between the column of the ground truth and the synthetic data.

**Score:** 0: no datatype mismatch. 1: complete data type mismatch between the datasets.

**static direction()** → str

**evaluate**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*: synthcity.plugins.core.dataloader.DataLoader) → Dict

**evaluate\_default**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*: synthcity.plugins.core.dataloader.DataLoader) → float

**classmethod fqdn()** → str

**static name()** → str

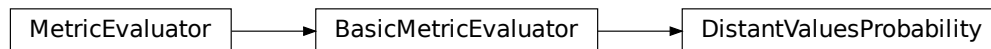
**reduction()** → Callable

**static type()** → str

**use\_cache**(*path*: pathlib.Path) → bool

**class DistantValuesProbability**(\*\*kwargs: Any)

Bases: [synthcity.metrics.eval\\_sanity.BasicMetricEvaluator](#)



Compute the probability of distant values between the real and synthetic data.

**Score:** 0 means there is no chance to have rows in the synthetic far away from the real data. 1 means all the synthetic datapoints are far away from the real data.

**static direction()** → str

**evaluate**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*: synthcity.plugins.core.dataloader.DataLoader) → Dict

**evaluate\_default**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*: synthcity.plugins.core.dataloader.DataLoader) → float

**classmethod fqdn()** → str

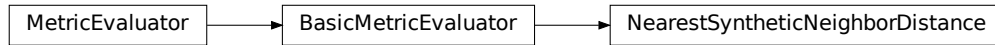
**static name()** → str

**reduction()** → Callable

**static type()** → str

**use\_cache**(*path*: pathlib.Path) → bool

```
class NearestSyntheticNeighborDistance(**kwargs: Any)
 Bases: synthcity.metrics.eval_sanity.BasicMetricEvaluator
```



Computes the <reduction>(distance) from the real data to the closest neighbor in the synthetic data

```
static direction() → str
```

```
evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → Dict
```

```
evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
```

```
classmethod fqdn() → str
```

```
static name() → str
```

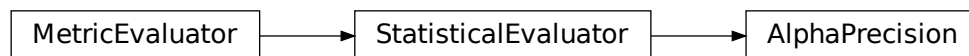
```
reduction() → Callable
```

```
static type() → str
```

```
use_cache(path: pathlib.Path) → bool
```

### 11.1.2 synthcity.metrics.eval\_statistical module

```
class AlphaPrecision(**kwargs: Any)
 Bases: synthcity.metrics.eval_statistical.StatisticalEvaluator
```



Evaluates the alpha-precision, beta-recall, and authenticity scores.

The class evaluates the synthetic data using a tuple of three metrics: alpha-precision, beta-recall, and authenticity. Note that these metrics can be evaluated for each synthetic data point (which are useful for auditing and post-processing). Here we average the scores to reflect the overall quality of the data. The formal definitions can be found in the reference below:

Alaa, Ahmed, Boris Van Breugel, Evgeny S. Saveliev, and Mihaela van der Schaar. “How faithful is your synthetic data? sample-level metrics for evaluating and auditing generative models.” In International Conference on Machine Learning, pp. 290-306. PMLR, 2022.

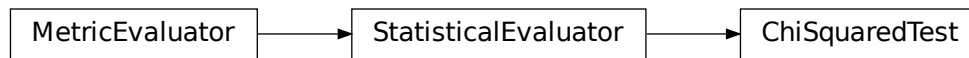
```
static direction() → str
```

```
evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → Dict
```

```

evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
classmethod fqdn() → str
metrics(X: numpy.ndarray, X_syn: numpy.ndarray, emb_center: Optional[numpy.ndarray] = None) →
 Tuple
static name() → str
reduction() → Callable
static type() → str
use_cache(path: pathlib.Path) → bool
class ChiSquaredTest(**kwargs: Any)
 Bases: synthcity.metrics.eval_statistical.StatisticalEvaluator

```



Performs the one-way chi-square test.

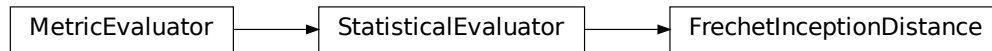
**Returns** The p-value. A small value indicates that we can reject the null hypothesis and that the distributions are different.

**Score:** 0: the distributions are different 1: the distributions are identical.

```

static direction() → str
evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → Dict
evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
classmethod fqdn() → str
static name() → str
reduction() → Callable
static type() → str
use_cache(path: pathlib.Path) → bool
class FrechetInceptionDistance(**kwargs: Any)
 Bases: synthcity.metrics.eval_statistical.StatisticalEvaluator

```



Calculates the Frechet Inception Distance (FID) to evaluate GANs.

Paper: GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium.

The FID metric calculates the distance between two distributions of images. Typically, we have summary statistics (mean & covariance matrix) of one of these distributions, while the 2nd distribution is given by a GAN.

Adapted by Boris van Breugel([bv292@cam.ac.uk](mailto:bv292@cam.ac.uk))

**static direction()** → str

**evaluate**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*:  
synthcity.plugins.core.dataloader.DataLoader) → Dict

**evaluate\_default**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*:  
synthcity.plugins.core.dataloader.DataLoader) → float

**classmethod fqdn()** → str

**static name()** → str

**reduction()** → Callable

**static type()** → str

**use\_cache**(*path*: pathlib.Path) → bool

**class InverseKLDivergence**(\*\*kwargs: Any)

Bases: *synthcity.metrics.eval\_statistical.StatisticalEvaluator*



Returns the average inverse of the Kullback–Leibler Divergence metric.

**Score:** 0: the datasets are from different distributions. 1: the datasets are from the same distribution.

**static direction()** → str

**evaluate**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*:  
synthcity.plugins.core.dataloader.DataLoader) → Dict

**evaluate\_default**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*:  
synthcity.plugins.core.dataloader.DataLoader) → float

**classmethod fqdn()** → str

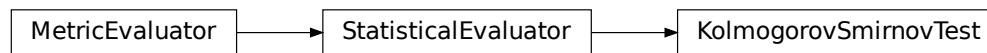
**static name()** → str

**reduction()** → Callable

```

 static type() → str
 use_cache(path: pathlib.Path) → bool
class JensenShannonDistance(normalize: bool = True, **kwargs: Any)
 Bases: synthcity.metrics.eval_statistical.StatisticalEvaluator
 Evaluate the average Jensen-Shannon distance (metric) between two probability arrays.
 static direction() → str
 evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → Dict
 evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
 classmethod fqdn() → str
 static name() → str
 reduction() → Callable
 static type() → str
 use_cache(path: pathlib.Path) → bool
class KolmogorovSmirnovTest(**kwargs: Any)
 Bases: synthcity.metrics.eval_statistical.StatisticalEvaluator

```



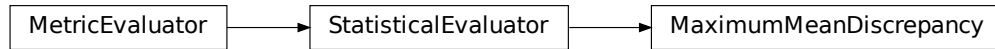
Performs the Kolmogorov-Smirnov test for goodness of fit.

**Score:** 0: the distributions are totally different. 1: the distributions are identical.

```

 static direction() → str
 evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → Dict
 evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
 classmethod fqdn() → str
 static name() → str
 reduction() → Callable
 static type() → str
 use_cache(path: pathlib.Path) → bool
class MaximumMeanDiscrepancy(kernel: str = 'rbf', **kwargs: Any)
 Bases: synthcity.metrics.eval_statistical.StatisticalEvaluator

```



Empirical maximum mean discrepancy. The lower the result the more evidence that distributions are the same.

**Parameters** **kernel** – “rbf”, “linear” or “polynomial”

**Score:** 0: The distributions are the same. 1: The distributions are totally different.

**static direction()** → str

**evaluate**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*:  
synthcity.plugins.core.dataloader.DataLoader) → Dict

**evaluate\_default**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*:  
synthcity.plugins.core.dataloader.DataLoader) → float

**classmethod fqdn()** → str

**static name()** → str

**reduction()** → Callable

**static type()** → str

**use\_cache**(*path*: pathlib.Path) → bool

**class PRDCScore**(*nearest\_k*: int = 5, *\*\*kwargs*: Any)  
Bases: [synthcity.metrics.eval\\_statistical.StatisticalEvaluator](#)



Computes precision, recall, density, and coverage given two manifolds.

**Parameters** **nearest\_k** – int.

**static direction()** → str

**evaluate**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*:  
synthcity.plugins.core.dataloader.DataLoader) → Dict

**evaluate\_default**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*:  
synthcity.plugins.core.dataloader.DataLoader) → float

**classmethod fqdn()** → str

**static name()** → str

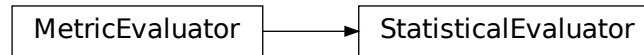
**reduction()** → Callable

**static type()** → str



```
use_cache(path: pathlib.Path) → bool
```

```
class StatisticalEvaluator(**kwargs: Any)
 Bases: synthcity.metrics.core.metric.MetricEvaluator
```



```
abstract static direction() → str
```

```
evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → Dict
```

```
evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
```

```
classmethod fqdn() → str
```

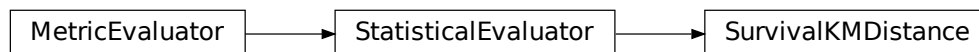
```
abstract static name() → str
```

```
reduction() → Callable
```

```
static type() → str
```

```
use_cache(path: pathlib.Path) → bool
```

```
class SurvivalKMDistance(**kwargs: Any)
 Bases: synthcity.metrics.eval_statistical.StatisticalEvaluator
```



The distance between two Kaplan-Meier plots. Used for survival analysis

```
static direction() → str
```

```
evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → Dict
```

```
evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
```

```
classmethod fqdn() → str
```

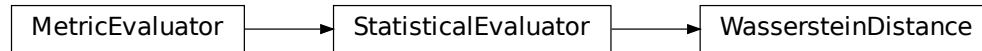
```
static name() → str
```

```
reduction() → Callable
```

```
static type() → str
```

```
use_cache(path: pathlib.Path) → bool
```

```
class WassersteinDistance(**kwargs: Any)
 Bases: synthcity.metrics.eval_statistical.StatisticalEvaluator
```



Compare Wasserstein distance between original data and synthetic data.

**Parameters**

- **X** – original data
- **X\_syn** – synthetically generated data

**Returns** Wasserstein distance

**Return type** WD\_value

**static direction()** → str

**evaluate**(X\_gt: *synthcity.plugins.core.dataloader.DataLoader*, X\_syn:  
*synthcity.plugins.core.dataloader.DataLoader*) → Dict

**evaluate\_default**(X\_gt: *synthcity.plugins.core.dataloader.DataLoader*, X\_syn:  
*synthcity.plugins.core.dataloader.DataLoader*) → float

**classmethod fqdn()** → str

**static name()** → str

**reduction()** → Callable

**static type()** → str

**use\_cache**(path: *pathlib.Path*) → bool

### 11.1.3 synthcity.metrics.eval\_performance module

```
class AugmentationPerformanceEvaluatorLinear(**kwargs: Any)
 Bases: synthcity.metrics.eval_performance.PerformanceEvaluatorLinear
```

**static direction()** → str

**evaluate**(X\_gt: *synthcity.plugins.core.dataloader.DataLoader*, X\_syn:  
*synthcity.plugins.core.dataloader.DataLoader*) → Dict

**evaluate\_default**(X\_gt: *synthcity.plugins.core.dataloader.DataLoader*, X\_syn:  
*synthcity.plugins.core.dataloader.DataLoader*) → float

**classmethod fqdn()** → str

**static name()** → str

**reduction()** → Callable

**static standard\_performance\_output\_keys()** → List

**static type()** → str

```

 use_cache(path: pathlib.Path) → bool

class AugmentationPerformanceEvaluatorMLP(**kwargs: Any)
 Bases: synthcity.metrics.eval_performance.PerformanceEvaluatorMLP

 static direction() → str

 evaluate(X_gt: synthcity.plugins.core.data_loader.DataLoader, X_syn:
 synthcity.plugins.core.data_loader.DataLoader) → Dict

 evaluate_default(X_gt: synthcity.plugins.core.data_loader.DataLoader, X_syn:
 synthcity.plugins.core.data_loader.DataLoader) → float

 classmethod fqdn() → str

 static name() → str

 reduction() → Callable

 static standard_performance_output_keys() → List

 static type() → str

 use_cache(path: pathlib.Path) → bool

class AugmentationPerformanceEvaluatorXGB(**kwargs: Any)
 Bases: synthcity.metrics.eval_performance.PerformanceEvaluatorXGB

 static direction() → str

 evaluate(X_gt: synthcity.plugins.core.data_loader.DataLoader, X_syn:
 synthcity.plugins.core.data_loader.DataLoader) → Dict

 evaluate_default(X_gt: synthcity.plugins.core.data_loader.DataLoader, X_syn:
 synthcity.plugins.core.data_loader.DataLoader) → float

 classmethod fqdn() → str

 static name() → str

 reduction() → Callable

 static standard_performance_output_keys() → List

 static type() → str

 use_cache(path: pathlib.Path) → bool

class FeatureImportanceRankDistance(distance: str = 'kendall', **kwargs: Any)
 Bases: synthcity.metrics.core.metric.MetricEvaluator

```

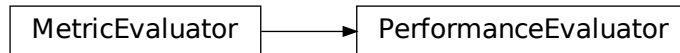


Train an XGBoost classifier or regressor on the synthetic data and evaluate the feature importance. Train an XGBoost model on the real data and evaluate the feature importance.

Returns the rank distance between the feature importance Returns the average performance discrepancy between training on real data vs on synthetic data.

**Score:** close to 1: similar performance close to 0: unrelated close to -1: the ranks have different monotony.

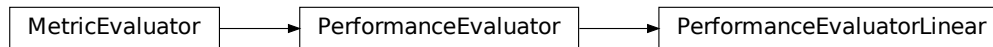
```
static direction() → str
distance(lhs: numpy.ndarray, rhs: numpy.ndarray) → Tuple[float, float]
evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → Dict
evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
classmethod fqdn() → str
static name() → str
reduction() → Callable
static type() → str
use_cache(path: pathlib.Path) → bool
class PerformanceEvaluator(**kwargs: Any)
 Bases: synthcity.metrics.core.metric.MetricEvaluator
```



Evaluating synthetic data based on downstream performance.

This implements the train-on-synthetic test-on-real methodology for evaluation.

```
static direction() → str
abstract evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → Dict
evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
classmethod fqdn() → str
abstract static name() → str
reduction() → Callable
static standard_performance_output_keys() → List
static type() → str
use_cache(path: pathlib.Path) → bool
class PerformanceEvaluatorLinear(**kwargs: Any)
 Bases: synthcity.metrics.eval_performance.PerformanceEvaluator
```



Train a Linear classifier or regressor on the synthetic data and evaluate the performance on real test data.

Returns the average performance discrepancy between training on real data vs on synthetic data.

**Score:** close to 1: similar performance close to 0: massive performance degradation

**static direction()** → str

**evaluate**(*X<sub>gt</sub>*: synthcity.plugins.core.data\_loader.DataLoader, *X<sub>syn</sub>*:  
synthcity.plugins.core.data\_loader.DataLoader) → Dict

**evaluate\_default**(*X<sub>gt</sub>*: synthcity.plugins.core.data\_loader.DataLoader, *X<sub>syn</sub>*:  
synthcity.plugins.core.data\_loader.DataLoader) → float

**classmethod fqdn()** → str

**static name()** → str

**reduction()** → Callable

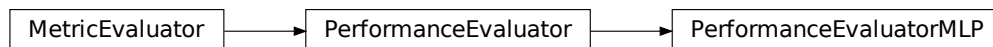
**static standard\_performance\_output\_keys()** → List

**static type()** → str

**use\_cache**(*path*: pathlib.Path) → bool

**class PerformanceEvaluatorMLP**(\*\*kwargs: Any)

Bases: [synthcity.metrics.eval\\_performance.PerformanceEvaluator](#)



Train a Neural Net classifier or regressor on the synthetic data and evaluate the performance on real test data.

Returns the average performance discrepancy between training on real data vs on synthetic data.

**Score:** close to 1: similar performance close to 1: massive performance degradation

**static direction()** → str

**evaluate**(*X<sub>gt</sub>*: synthcity.plugins.core.data\_loader.DataLoader, *X<sub>syn</sub>*:  
synthcity.plugins.core.data\_loader.DataLoader) → Dict

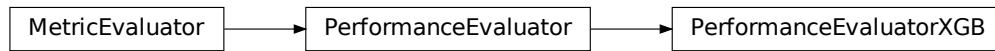
**evaluate\_default**(*X<sub>gt</sub>*: synthcity.plugins.core.data\_loader.DataLoader, *X<sub>syn</sub>*:  
synthcity.plugins.core.data\_loader.DataLoader) → float

**classmethod fqdn()** → str

**static name()** → str

**reduction()** → Callable

```
static standard_performance_output_keys() → List
static type() → str
use_cache(path: pathlib.Path) → bool
class PerformanceEvaluatorXGB(**kwargs: Any)
 Bases: synthcity.metrics.eval_performance.PerformanceEvaluator
```



Train an XGBoost classifier or regressor on the synthetic data and evaluate the performance on real test data.

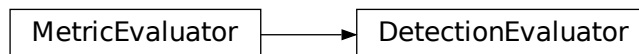
Returns the average performance discrepancy between training on real data vs on synthetic data.

**Score:** close to 1: similar performance close to 0: massive performance degradation

```
static direction() → str
evaluate(X_gt: synthcity.plugins.core.data_loader.DataLoader, X_syn:
 synthcity.plugins.core.data_loader.DataLoader) → Dict
evaluate_default(X_gt: synthcity.plugins.core.data_loader.DataLoader, X_syn:
 synthcity.plugins.core.data_loader.DataLoader) → float
classmethod fqdn() → str
static name() → str
reduction() → Callable
static standard_performance_output_keys() → List
static type() → str
use_cache(path: pathlib.Path) → bool
```

#### 11.1.4 synthcity.metrics.eval\_detection module

```
class DetectionEvaluator(**kwargs: Any)
 Bases: synthcity.metrics.core.metric.MetricEvaluator
```



Train a SKLearn classifier to detect the synthetic data from real data.

Synthetic and real data are combined to form a new dataset. K-fold cross validation is performed to see how well a classifier can distinguish real from synthetic.

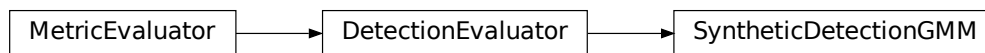
**Returns** The average AUCROC score for detecting synthetic data.

**Score:** 0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.

```

static direction() → str
evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → Dict
evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
classmethod fqdn() → str
static name() → str
reduction() → Callable
static type() → str
use_cache(path: pathlib.Path) → bool
class SyntheticDetectionGMM(**kwargs: Any)
 Bases: synthcity.metrics.eval_detection.DetectionEvaluator

```



Train a GaussianMixture model to detect synthetic data.

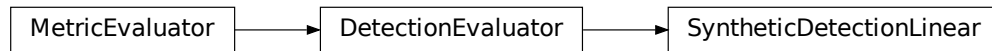
**Returns** The average score for detecting synthetic data.

**Score:** 0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.

```

static direction() → str
evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → Dict
evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
classmethod fqdn() → str
static name() → str
reduction() → Callable
static type() → str
use_cache(path: pathlib.Path) → bool
class SyntheticDetectionLinear(**kwargs: Any)
 Bases: synthcity.metrics.eval_detection.DetectionEvaluator

```



Train a LogisticRegression classifier to detect the synthetic data.

**Returns** The average AUCROC score for detecting synthetic data.

**Score:** 0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.

**static direction()** → str

**evaluate**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*:  
synthcity.plugins.core.dataloader.DataLoader) → Dict

**evaluate\_default**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*:  
synthcity.plugins.core.dataloader.DataLoader) → float

**classmethod fqdn()** → str

**static name()** → str

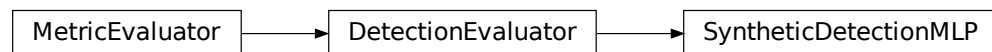
**reduction()** → Callable

**static type()** → str

**use\_cache**(*path*: pathlib.Path) → bool

**class SyntheticDetectionMLP**(\*\*kwargs: Any)

Bases: [synthcity.metrics.eval\\_detection.DetectionEvaluator](#)



Train a MLP classifier to detect the synthetic data.

**Returns** The average AUCROC score for detecting synthetic data.

**Score:** 0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.

**static direction()** → str

**evaluate**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*:  
synthcity.plugins.core.dataloader.DataLoader) → Dict

**evaluate\_default**(*X\_gt*: synthcity.plugins.core.dataloader.DataLoader, *X\_syn*:  
synthcity.plugins.core.dataloader.DataLoader) → float

**classmethod fqdn()** → str

**static name()** → str

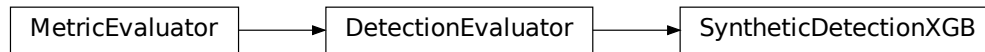
**reduction()** → Callable



```

static type() → str
use_cache(path: pathlib.Path) → bool
class SyntheticDetectionXGB(**kwargs: Any)
 Bases: synthcity.metrics.eval_detection.DetectionEvaluator

```



Train a XGBoostclassifier to detect the synthetic data.

**Returns** The average AUCROC score for detecting synthetic data.

**Score:** 0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.

```

static direction() → str
evaluate(X_gt: synthcity.plugins.core.data_loader.DataLoader, X_syn:
 synthcity.plugins.core.data_loader.DataLoader) → Dict
evaluate_default(X_gt: synthcity.plugins.core.data_loader.DataLoader, X_syn:
 synthcity.plugins.core.data_loader.DataLoader) → float
classmethod fqdn() → str
static name() → str
reduction() → Callable
static type() → str
use_cache(path: pathlib.Path) → bool

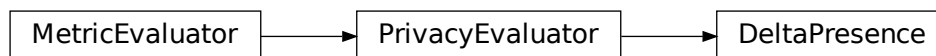
```

### 11.1.5 synthcity.metrics.eval\_privacy module

```

class DeltaPresence(**kwargs: Any)
 Bases: synthcity.metrics.eval_privacy.PrivacyEvaluator

```



Returns the maximum re-identification probability on the real dataset from the synthetic dataset.

For each dataset partition, we report the maximum ratio of unique sensitive information between the real dataset and in the synthetic dataset.

```

static direction() → str

```

```
evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader, *args: Any, **kwargs: Any) → Dict
evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
classmethod fqdn() → str
static name() → str
reduction() → Callable
static type() → str
use_cache(path: pathlib.Path) → bool

class DomiasMIA(**kwargs: Any)
 Bases: synthcity.metrics.eval_privacy.PrivacyEvaluator

 DOMIAS is a membership inference attacker model against synthetic data, that incorporates density estimation
 to detect generative model overfitting. That is it uses local overfitting to detect whether a data point was used to
 train the generative model or not.

 Returns: A dictionary with a key for each of the synthetic_sizes values. For each synthetic_sizes value, the
 dictionary contains the keys:

 • MIA_performance : accuracy and AUCROC for each attack

 • MIA_scores: output scores for each attack

 Reference: Boris van Breugel, Hao Sun, Zhaozhi Qian, Mihaela van der Schaar, AISTATS 2023. DOMIAS:
 Membership Inference Attacks against Synthetic Data through Overfitting Detection.

 static direction() → str
evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader, *args: Any, **kwargs: Any) → Dict
evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader, X_train:
 synthcity.plugins.core.dataloader.DataLoader, X_ref_syn:
 synthcity.plugins.core.dataloader.DataLoader, reference_size: int) → float
abstract evaluate_p_R(synth_set: Union[synthcity.plugins.core.dataloader.DataLoader, Any],
 synth_val_set: Union[synthcity.plugins.core.dataloader.DataLoader, Any],
 reference_set: numpy.ndarray, X_test: numpy.ndarray, device: Any) → Any
classmethod fqdn() → str
static name() → str
reduction() → Callable
static type() → str
use_cache(path: pathlib.Path) → bool

class DomiasMIABNAF(**kwargs: Any)
 Bases: synthcity.metrics.eval_privacy.DomiasMIA

 static direction() → str
evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader, *args: Any, **kwargs: Any) → Dict
```

```

evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader, X_train:
 synthcity.plugins.core.dataloader.DataLoader, X_ref_syn:
 synthcity.plugins.core.dataloader.DataLoader, reference_size: int) → float

evaluate_p_R(synth_set: Union[synthcity.plugins.core.dataloader.DataLoader, Any], synth_val_set:
 Union[synthcity.plugins.core.dataloader.DataLoader, Any], reference_set: numpy.ndarray,
 X_test: numpy.ndarray, device: Any) → Tuple[numpy.ndarray, numpy.ndarray]

classmethod fqdn() → str

static name() → str

reduction() → Callable

static type() → str

use_cache(path: pathlib.Path) → bool

class DomiasMIAKDE(**kwargs: Any)
 Bases: synthcity.metrics.eval_privacy.DomiasMIA

 static direction() → str

 evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader, *args: Any, **kwargs: Any) → Dict

 evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader, X_train:
 synthcity.plugins.core.dataloader.DataLoader, X_ref_syn:
 synthcity.plugins.core.dataloader.DataLoader, reference_size: int) → float

 evaluate_p_R(synth_set: Union[synthcity.plugins.core.dataloader.DataLoader, Any], synth_val_set:
 Union[synthcity.plugins.core.dataloader.DataLoader, Any], reference_set: numpy.ndarray,
 X_test: numpy.ndarray, device: Any) → Tuple[numpy.ndarray, numpy.ndarray]

 classmethod fqdn() → str

 static name() → str

 reduction() → Callable

 static type() → str

 use_cache(path: pathlib.Path) → bool

class DomiasMIAPrior(**kwargs: Any)
 Bases: synthcity.metrics.eval_privacy.DomiasMIA

 static direction() → str

 evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader, *args: Any, **kwargs: Any) → Dict

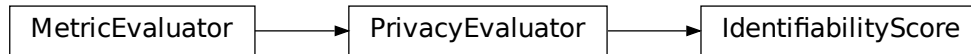
 evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader, X_train:
 synthcity.plugins.core.dataloader.DataLoader, X_ref_syn:
 synthcity.plugins.core.dataloader.DataLoader, reference_size: int) → float

 evaluate_p_R(synth_set: Union[synthcity.plugins.core.dataloader.DataLoader, Any], synth_val_set:
 Union[synthcity.plugins.core.dataloader.DataLoader, Any], reference_set: numpy.ndarray,
 X_test: numpy.ndarray, device: Any) → Tuple[numpy.ndarray, numpy.ndarray]

 classmethod fqdn() → str

```

```
static name() → str
reduction() → Callable
static type() → str
use_cache(path: pathlib.Path) → bool
class IdentifiabilityScore(**kwargs: Any)
 Bases: synthcity.metrics.eval_privacy.PrivacyEvaluator
```



Returns the re-identification score on the real dataset from the synthetic dataset.

We estimate the risk of re-identifying any real data point using synthetic data. Intuitively, if the synthetic data are very close to the real data, the re-identification risk would be high. The precise formulation of the re-identification score is given in the reference below.

Reference: Jinsung Yoon, Lydia N. Drumright, Mihaela van der Schaar, “Anonymization through Data Synthesis using Generative Adversarial Networks (ADS-GAN): A harmonizing advancement for AI in medicine,” IEEE Journal of Biomedical and Health Informatics (JBHI), 2019. Paper link: <https://ieeexplore.ieee.org/document/9034117>

```
static direction() → str
evaluate(X_gt: synthcity.plugins.core.data_loader.DataLoader, X_syn:
 synthcity.plugins.core.data_loader.DataLoader, *args: Any, **kwargs: Any) → Dict
evaluate_default(X_gt: synthcity.plugins.core.data_loader.DataLoader, X_syn:
 synthcity.plugins.core.data_loader.DataLoader) → float
classmethod fqdn() → str
static name() → str
reduction() → Callable
static type() → str
use_cache(path: pathlib.Path) → bool
class PrivacyEvaluator(**kwargs: Any)
 Bases: synthcity.metrics.core.metric.MetricEvaluator
```

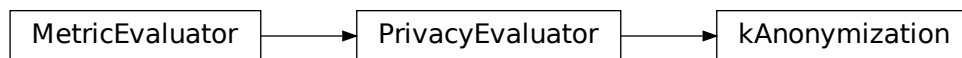


```
abstract static direction() → str
```

```

evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader, *args: Any, **kwargs: Any) → Dict
evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
classmethod fqdn() → str
abstract static name() → str
reduction() → Callable
static type() → str
use_cache(path: pathlib.Path) → bool
class kAnonymization(**kwargs: Any)
 Bases: synthcity.metrics.eval_privacy.PrivacyEvaluator

```



Returns the k-anon ratio between the real data and the synthetic data. For each dataset, it is computed the value k which satisfies the k-anonymity rule: each record is similar to at least another k-1 other records on the potentially identifying variables.

```

static direction() → str
evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader, *args: Any, **kwargs: Any) → Dict
evaluate_data(X: synthcity.plugins.core.dataloader.DataLoader) → int
evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
classmethod fqdn() → str
static name() → str
reduction() → Callable
static type() → str
use_cache(path: pathlib.Path) → bool
class kMap(**kwargs: Any)
 Bases: synthcity.metrics.eval_privacy.PrivacyEvaluator

```



Returns the minimum value  $k$  that satisfies the  $k$ -map rule.

The data satisfies  $k$ -map if every combination of values for the quasi-identifiers appears at least  $k$  times in the reidentification(synthetic) dataset.

**static direction()**  $\rightarrow$  str

**evaluate**( $X_{gt}$ : synthcity.plugins.core.dataloader.DataLoader,  $X_{syn}$ :  
synthcity.plugins.core.dataloader.DataLoader,  $*args$ : Any,  $**kwargs$ : Any)  $\rightarrow$  Dict

**evaluate\_default**( $X_{gt}$ : synthcity.plugins.core.dataloader.DataLoader,  $X_{syn}$ :  
synthcity.plugins.core.dataloader.DataLoader)  $\rightarrow$  float

**classmethod fqdn()**  $\rightarrow$  str

**static name()**  $\rightarrow$  str

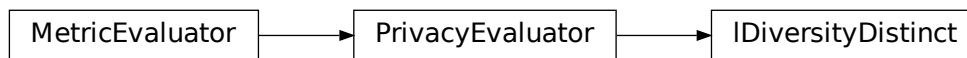
**reduction()**  $\rightarrow$  Callable

**static type()**  $\rightarrow$  str

**use\_cache**( $path$ : pathlib.Path)  $\rightarrow$  bool

**class lDiversityDistinct**( $**kwargs$ : Any)

Bases: [synthcity.metrics.eval\\_privacy.PrivacyEvaluator](#)



Returns the distinct  $l$ -diversity ratio between the real data and the synthetic data.

For each dataset, it computes the minimum value  $l$  which satisfies the distinct  $l$ -diversity rule: every generalized block has to contain at least  $l$  different sensitive values.

We simulate a set of the cluster over the dataset, and we return the minimum length of unique sensitive values for any cluster.

**static direction()**  $\rightarrow$  str

**evaluate**( $X_{gt}$ : synthcity.plugins.core.dataloader.DataLoader,  $X_{syn}$ :  
synthcity.plugins.core.dataloader.DataLoader,  $*args$ : Any,  $**kwargs$ : Any)  $\rightarrow$  Dict

**evaluate\_data**( $X$ : synthcity.plugins.core.dataloader.DataLoader)  $\rightarrow$  int

**evaluate\_default**( $X_{gt}$ : synthcity.plugins.core.dataloader.DataLoader,  $X_{syn}$ :  
synthcity.plugins.core.dataloader.DataLoader)  $\rightarrow$  float

**classmethod fqdn()**  $\rightarrow$  str

**static name()**  $\rightarrow$  str

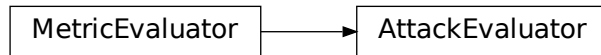
**reduction()**  $\rightarrow$  Callable

**static type()**  $\rightarrow$  str

**use\_cache**( $path$ : pathlib.Path)  $\rightarrow$  bool

### 11.1.6 synthcity.metrics.eval\_attacks module

```
class AttackEvaluator(**kwargs: Any)
 Bases: synthcity.metrics.core.metric.MetricEvaluator
```



Evaluating the risk of attribute inference attack.

This class evaluates the risk of a type of privacy attack, known as attribute inference attack. In this setting, the attacker has access to the synthetic dataset as well as partial information about the real data (quasi-identifiers). The attacker seeks to uncover the sensitive attributes of the real data using these two pieces of information.

```
abstract static direction() → str
```

```
abstract evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → Dict
```

```
evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
```

```
classmethod fqdn() → str
```

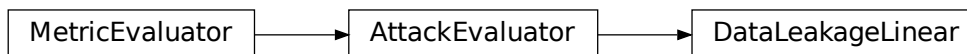
```
abstract static name() → str
```

```
reduction() → Callable
```

```
static type() → str
```

```
use_cache(path: pathlib.Path) → bool
```

```
class DataLeakageLinear(**kwargs: Any)
 Bases: synthcity.metrics.eval_attacks.AttackEvaluator
```



Data leakage test using a linear model

```
static direction() → str
```

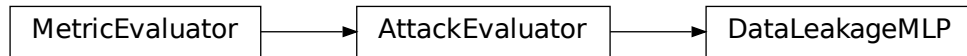
```
evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → Dict
```

```
evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
```

```
classmethod fqdn() → str
```

```
static name() → str
reduction() → Callable
static type() → str
use_cache(path: pathlib.Path) → bool

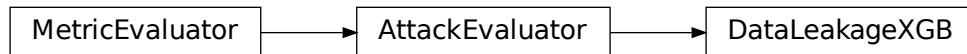
class DataLeakageMLP(**kwargs: Any)
 Bases: synthcity.metrics.eval_attacks.AttackEvaluator
```



Data leakage test using a neural net.

```
static direction() → str
evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → Dict
evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
classmethod fqdn() → str
static name() → str
reduction() → Callable
static type() → str
use_cache(path: pathlib.Path) → bool

class DataLeakageXGB(**kwargs: Any)
 Bases: synthcity.metrics.eval_attacks.AttackEvaluator
```



Data leakage test using XGBoost

```
static direction() → str
evaluate(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → Dict
evaluate_default(X_gt: synthcity.plugins.core.dataloader.DataLoader, X_syn:
 synthcity.plugins.core.dataloader.DataLoader) → float
classmethod fqdn() → str
```



```

static name() → str
reduction() → Callable
static type() → str
use_cache(path: pathlib.Path) → bool

```

### 11.1.7 synthcity.metrics.weighted\_metrics module

```

class WeightedMetrics(metrics: List[Tuple[str, str]], weights: List[float], task_type: str = 'classification',
 random_state: int = 0, workspace: pathlib.Path = PosixPath('workspace'))
 Bases: object
 direction() → str
 evaluate(X_gt: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame],
 X_syn: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame]) →
 float

```

### 11.1.8 synthcity.benchmark package

```

class Benchmarks
 Bases: object
 static evaluate(tests: List[Tuple[str, str, dict]], X: synthcity.plugins.core.dataloader.DataLoader, X_test:
 Optional[synthcity.plugins.core.dataloader.DataLoader] = None, metrics: Optional[Dict]
 = None, repeats: int = 3, synthetic_size: Optional[int] = None, synthetic_constraints:
 Optional[synthcity.plugins.core.constraints.Constraints] = None, synthetic_cache: bool =
 True, synthetic_reuse_if_exists: bool = True, augmented_reuse_if_exists: bool = True,
 task_type: str = 'classification', workspace: pathlib.Path = PosixPath('workspace'),
 augmentation_rule: str = 'equal', strict_augmentation: bool = False,
 ad_hoc_augment_vals: Optional[Dict] = None, use_metric_cache: bool = True,
 **generate_kwargs: Any) → pandas.core.frame.DataFrame
 Benchmark the performance of several algorithms.

```

#### Parameters

- **tests** – List[Tuple[str, str, dict]] Tuples of form (testname: str, plugin\_name, str, plugin\_args: dict)
- **X** – DataLoader The baseline dataset to learn
- **X\_test** – Optional[DataLoader] Optional test dataset for evaluation. If None, X will be split in train/test datasets.
- **metrics** – List of metrics to test. By default, all metrics are evaluated. Full dictionary of metrics is: {
   
 'sanity': ['data\_mismatch', 'common\_rows\_proportion', 'nearest\_syn\_neighbor\_distance', 'close\_values\_probability', 'distant\_values\_probability'], 'stats': ['jensenshannon\_dist', 'chi\_squared\_test', 'feature\_corr', 'inv\_kl\_divergence', 'ks\_test', 'max\_mean\_discrepancy', 'wasserstein\_dist', 'prdc', 'alpha\_precision', 'survival\_km\_distance'], 'performance': ['linear\_model', 'mlp', 'xgb', 'feat\_rank\_distance'], 'detection': ['detection\_xgb', 'detection\_mlp', 'detection\_gmm', 'detection\_linear'], 'privacy': ['delta-presence', 'k-anonymization', 'k-map', 'distinct l-diversity'],

```
 'identifiability_score', 'DomiasMIA_BNAF', 'DomiasMIA_KDE', 'Domias-
 MIA_prior']
 }
```

- **repeats** – Number of test repeats
- **synthetic\_size** – int The size of the synthetic dataset. By default, it is len(X).
- **synthetic\_constraints** – Optional constraints on the synthetic data. By default, it inherits the constraints from X.
- **synthetic\_cache** – bool Enable experiment caching
- **synthetic\_reuse\_if\_exists** – bool If the current synthetic dataset is cached, it will be reused for the experiments. Defaults to True.
- **augmented\_reuse\_if\_exists** – bool If the current augmented dataset is cached, it will be reused for the experiments. Defaults to True.
- **task\_type** – str The type of problem. Relevant for evaluating the downstream models with the correct metrics. Valid tasks are: “classification”, “regression”, “survival\_analysis”, “time\_series”, “time\_series\_survival”.
- **workspace** – Path Path for caching experiments. Default: “workspace”.
- **augmentation\_rule** – str The rule used to achieve the desired proportion records with each value in the fairness column. Possible values are: ‘equal’, ‘log’, and ‘ad-hoc’. Defaults to “equal”.
- **strict\_augmentation** – bool Flag to ensure that the condition for generating synthetic data is strictly met. Defaults to False.
- **ad\_hoc\_augment\_vals** – Dict A dictionary containing the number of each class to augment the real data with. This is only required if using the rule=”ad-hoc” option. Defaults to None.
- **use\_metric\_cache** – bool If the current metric has been previously run and is cached, it will be reused for the experiments. Defaults to True.
- **plugin\_kwargs** – Optional kwargs for each algorithm. Example {“adsgan”: {“n\_iter”: 10}},

**static highlight**(results: Dict) → None

**static print**(results: Dict, only\_comparatives: bool = True) → None

**print\_score**(mean: pandas.core.series.Series, std: pandas.core.series.Series) → pandas.core.series.Series

## Submodules

### synthcity.benchmark.utils module

**augment\_data**(X\_train: synthcity.plugins.core.dataloader.DataLoader, augment\_generator: Any, strict: bool = False, rule: typing\_extensions.Literal[equal, log, ad - hoc] = 'equal', ad\_hoc\_augment\_vals: Optional[Dict[Any, int]] = None, synthetic\_constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, \*\*generate\_kwargs: Any) → synthcity.plugins.core.dataloader.DataLoader

Augment the real data with generated synthetic data

#### Parameters

- **X\_train** (`DataLoader`) – The ground truth `DataLoader` to augment with synthetic data.
- **augment\_generator** (`Any`) – The synthetic model to be used to generate the synthetic portion of the augmented dataset.
- **strict** (`bool`, *optional*) – Flag to ensure that the condition for generating synthetic data is strictly met. Defaults to `False`.
- **rule** (`Literal["equal", "log", "ad-hoc"]`) – The rule used to achieve the desired proportion records with each value in the fairness column. Defaults to `"equal"`.
- **ad\_hoc\_augment\_vals** (`Dict[Union[int, str], int]`, *optional*) – A dictionary containing the number of each class to augment the real data with. This is only required if using the rule=`"ad-hoc"` option. Defaults to `None`.
- **synthetic\_constraints** (`Optional[Constraints]`) – Constraints placed on the generation of the synthetic data. Defaults to `None`.

**Returns** The augmented dataset and labels.

**Return type** `DataLoader`

**calculate\_fair\_aug\_sample\_size**(`X_train: pandas.core.frame.DataFrame`, `fairness_column: Optional[str]`, `rule: typing_extensions.Literal[equal, log, ad - hoc]`, `ad_hoc_augment_vals: Optional[Dict[Any, int]] = None`) → `Dict`

Calculate how many samples to augment.

**Parameters**

- **X\_train** (`pd.DataFrame`) – The real dataset to be augmented.
- **fairness\_column** (`str`) – The column name of the column to test the fairness of a downstream model with respect to.
- **rule** (`Literal["equal", "log", "ad-hoc"]`) – The rule used to achieve the desired proportion records with each value in the fairness column. Defaults to `"equal"`.
- **ad\_hoc\_augment\_vals** (`Dict[ Union[int, str], int ]`, *optional*) – A dictionary containing the number of each class to augment the real data with. If using rule=`"ad-hoc"` this function returns `ad_hoc_augment_vals`, otherwise this parameter is ignored. Defaults to `{}`.

**Returns** A dictionary containing the number of each class to augment the real data with.

**Return type** `Dict`

**get\_json\_serializable\_kwargs**(`kwargs: Dict`) → `Dict`

This function should take the kwargs for `Benchmarks.evaluate` and makes them serializable with `json.dumps`. Currently it only handles `pathlib.Path` -> `str`.



## ADVANCED TOPICS

### 12.1 Advanced topics

#### 12.1.1 Core components

##### `synthcity.plugins.core.plugin` module

```
class Plugin(sampling_patience: int = 500, strict: bool = True, device: Any = device(type='cpu'), random_state:
 int = 0, workspace: pathlib.Path = PosixPath('workspace'), compress_dataset: bool = False,
 sampling_strategy: str = 'marginal')
```

Bases: `synthcity.plugins.core.serializable.Serializable`



Base class for all plugins.

**Each derived class must implement the following methods:** `type()` - a static method that returns the type of the plugin. e.g., `debug`, `generative`, `bayesian`, etc. `name()` - a static method that returns the name of the plugin. e.g., `ctgan`, `random_noise`, etc. `hyperparameter_space()` - a static method that returns the hyperparameters that can be tuned during AutoML. `_fit()` - internal method, called by *fit* on each training set. `_generate()` - internal method, called by *generate*.

If any method implementation is missing, the class constructor will fail.

##### Parameters

- **strict** – bool. Default = True If True, it raises an exception if the generated data is not following the requested constraints. If False, it returns only the rows that match the constraints.
- **workspace** – Path Path for caching intermediary results
- **compress\_dataset** – bool. Default = False Drop redundant features before training the generator.
- **device** – PyTorch device: `cpu` or `cuda`.
- **random\_state** – int Random seed

- **sampling\_patience** – int. Max inference iterations to wait for the generated data to match the training schema.
- **sampling\_strategy** – str Internal parameter for schema. marginal or uniform.

**class Config**

Bases: object

**arbitrary\_types\_allowed** = True

**validate\_assignment** = True

**fit**(X: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame], \*args: Any, \*\*kwargs: Any) → Any

Training method the synthetic data plugin.

#### Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import _
↳GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> #
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>> rules=[
>>> ("target", "==", 1),
>>>]
>>>)
>>> X_gen = test_plugin.generate(count,
>>> cond=np.ones(count),
>>> constraints=constraints
>>>)
>>> assert (X_gen["target"] == 1).all()
```

**Returns** self

**classmethod fqdn()** → str

The Fully-Qualified name of the plugin.

**generate**(*count: Optional[int] = None, constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None, random\_state: Optional[int] = None, \*\*kwargs: Any*) → [synthcity.plugins.core.data\\_loader.DataLoader](#)

Synthetic data generation method.

#### Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference\_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

#### Valid Operations:

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

#### Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>> rules=[
>>> ("InterestingFeature", "=", 0),
>>>]
>>>)
>>>
>>> syn_data = syn_model.generate(
>>> count=count,
>>> constraints=constraints
>>>).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random\_state** – optional int. Optional random seed to use.

**Returns** <count> synthetic samples

**abstract static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**abstract static name**() → str  
The name of the plugin.

**plot**(plt: Any, X: synthcity.plugins.core.data\_loader.DataLoader, count: Optional[int] = None, plots: list = ['marginal', 'associations', 'tsne'], \*\*kwargs: Any) → Any  
Plot the real-synthetic distributions.

#### Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

**Returns** self

**classmethod sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**classmethod sample\_hyperparameters\_optuna**(trial: Any, \*args: Any, \*\*kwargs: Any) → Dict[str, Any]

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(path: pathlib.Path) → bytes

**schema**() → synthcity.plugins.core.schema.Schema  
The reference schema

**schema\_includes**(other: Union[synthcity.plugins.core.data\_loader.DataLoader, pandas.core.frame.DataFrame]) → bool  
Helper method to test if the reference schema includes a Dataset

**Parameters** **other** – DataLoader. The dataset to test

**Returns** bool, if the schema includes the dataset or not.

**training\_schema**() → synthcity.plugins.core.schema.Schema  
The internal schema

**abstract static type**() → str  
The type of the plugin.

**static version**() → str  
API version

**class PluginLoader**(plugins: list, expected\_type: Type, categories: list)  
Bases: object

Plugin loading utility class. Used to load the plugins from the current folder.

**add**(name: str, cls: Type) → synthcity.plugins.core.plugin.PluginLoader  
Add a new plugin

**get**(name: str, \*args: Any, \*\*kwargs: Any) → Any  
Create a new object from a plugin. :param name: str. The name of the plugin :param &args: :param \*\*kwargs. Plugin specific arguments:



**Returns** The new object

**get\_type**(*name: str*) → Type

Get the class type of a plugin. :param name: str. The name of the plugin

**Returns** The class of the plugin

**list**() → List[str]

Get all the available plugins.

**load**(*buff: bytes*) → Any

Load serialized plugin

**reload**() → *synthcity.plugins.core.plugin.PluginLoader*

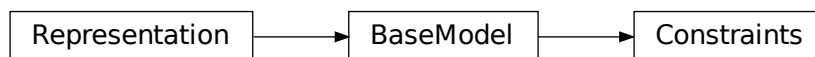
**types**() → List[Type]

Get the loaded plugins types

## synthcity.plugins.core.constraints module

**class Constraints**(\*, *rules: list = []*)

Bases: *pydantic.main.BaseModel*



Constraints on data.

The Constraints class allows users to specify constraints on the features. Examples include the feature value range, allowed item set, and data type. These constraints can be used to filter out invalid values in synthetic datasets.

### Constructor Args:

**rules: List[Tuple]** Each tuple in the list specifies a constraint on a feature. The tuple has the form of (feature, op, thresh), where feature is the feature name to apply constraint on, op takes values in [

“<”, “>=”, “<=”, “>”, “==”, “lt”, “le”, “gt”, “ge”, “eq”, “in”, “dtype”,

],

and thresh is the threshold or data type.

### Config

alias of *pydantic.config.BaseConfig*

**classmethod construct**(*\_fields\_set: Optional[SetStr] = None, \*\*values: Any*) → Model

Creates a new model setting `__dict__` and `__fields_set__` from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra* = ‘allow’ was set since it adds all passed values

**copy**(\*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude:*

*Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, update: Optional[DictStrAny] = None, deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

**Parameters**

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

**Returns** new model instance

**dict**(\*, include: *Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, exclude: *Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, by\_alias: *bool = False*, skip\_defaults: *Optional[bool] = None*, exclude\_unset: *bool = False*, exclude\_defaults: *bool = False*, exclude\_none: *bool = False*) → DictStrAny  
Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

**extend**(other: [synthcity.plugins.core.constraints.Constraints](#)) → [synthcity.plugins.core.constraints.Constraints](#)  
Extend the local constraints with more constraints.

**Parameters** **other** – The new constraints to add.

**Returns** self with the updated constraints.

**feature\_constraints**(ref\_feature: str) → List  
Get constraints for a given feature

**Parameters** **ref\_feature** – str The name of the feature of interest.

**Returns**

[('le', 3.), ('gt', 1.)]

If ref\_feature has no constraint, None will be returned.

**Return type** A list of tuples of (op, threshold) For example

**feature\_params**(feature: str) → Tuple  
Provide the parameters of Distribution from the Constraint  
This is to be used with the constraint\_to\_distribution function in distribution module.

**Parameters** **feature** – str The name of the feature of interest.

**Returns**

**str** The type of inferred distribution from (“categorical”, “float”, “integer”)

**dist\_args: Dict** The arguments to the constructor of the Distribution.

**Return type** dist\_template

**features**() → List  
Return list of feature names in an undefined order

**filter**(X: *pandas.core.frame.DataFrame*) → *pandas.core.frame.DataFrame*  
Apply the constraints to a DataFrame X.

**Parameters** **X** – DataFrame. The dataset to apply the constraints on.

**Returns** pandas.Index which matches all the constraints

**classmethod from\_orm**(obj: Any) → Model

**is\_valid**(X: *pandas.core.frame.DataFrame*) → bool

Checks if all the rows in X meet the constraints.

**Parameters** **X** – DataFrame. The dataset to apply the constraints on.

**Returns** True if all rows match the constraints, False otherwise

**json**(\*, include: *Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, exclude: *Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, by\_alias: *bool = False*, skip\_defaults: *Optional[bool] = None*, exclude\_unset: *bool = False*, exclude\_defaults: *bool = False*, exclude\_none: *bool = False*, encoder: *Optional[Callable[[Any], Any]] = None*, models\_as\_dict: *bool = True*, \*\**dumps\_kwargs*: *Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

*encoder* is an optional function to supply as *default* to *json.dumps()*, other arguments as per *json.dumps()*.

**match**(X: *pandas.core.frame.DataFrame*) → *pandas.core.frame.DataFrame*

Apply the constraints to a DataFrame X and return the filtered dataset.

**Parameters** **X** – DataFrame. The dataset to apply the constraints on.

**Returns** The filtered Dataframe

**classmethod parse\_file**(path: *Union[str, pathlib.Path]*, \*, content\_type: *unicode = None*, encoding: *unicode = 'utf8'*, proto: *pydantic.parse.Protocol = None*, allow\_pickle: *bool = False*) → Model

**classmethod parse\_obj**(obj: *Any*) → Model

**classmethod parse\_raw**(b: *Union[str, bytes]*, \*, content\_type: *unicode = None*, encoding: *unicode = 'utf8'*, proto: *pydantic.parse.Protocol = None*, allow\_pickle: *bool = False*) → Model

**rules**: list

**classmethod schema**(by\_alias: *bool = True*, ref\_template: *unicode = '#/definitions/{model}'*) → DictStrAny

**classmethod schema\_json**(\* , by\_alias: *bool = True*, ref\_template: *unicode = '#/definitions/{model}'*, \*\**dumps\_kwargs*: *Any*) → unicode

**classmethod update\_forward\_refs**(\*\**localns*: *Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

**classmethod validate**(value: *Any*) → Model

## synthcity.plugins.core.distribution module

**class CategoricalDistribution**(\* , name: *str*, data: *Optional[pandas.core.series.Series] = None*, random\_state: *int = 0*, marginal\_distribution: *Optional[pandas.core.series.Series] = None*, choices: *list = []*)

Bases: [synthcity.plugins.core.distribution.Distribution](#)



**class Config**

Bases: object

**arbitrary\_types\_allowed = True**

**as\_constraint()** → [synthcity.plugins.core.constraints.Constraints](#)

Convert the Distribution to a set of Constraints.

**choices:** list

**classmethod construct**(*\_fields\_set: Optional[SetStr] = None, \*\*values: Any*) → Model

Creates a new model setting `__dict__` and `__fields_set__` from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if `Config.extra = 'allow'` was set since it adds all passed values

**copy**(*\*, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, update: Optional[DictStrAny] = None, deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

#### Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

**Returns** new model instance

**data:** Optional[pandas.core.series.Series]

**dict**(*\*, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude:*

*Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, by\_alias: bool = False, skip\_defaults:*

*Optional[bool] = None, exclude\_unset: bool = False, exclude\_defaults: bool = False, exclude\_none:*

*bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

**dtype()** → str

**classmethod from\_orm**(*obj: Any*) → Model

**get()** → List[Any]

Return the metadata of the Distribution.

**has**(*val: Any*) → bool

Test if a value is included in the Distribution.

**includes**(*other: [synthcity.plugins.core.distribution.Distribution](#)*) → bool

Test if another Distribution is included in the local one.

**json**(*\*, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude:*

*Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, by\_alias: bool = False, skip\_defaults:*

*Optional[bool] = None, exclude\_unset: bool = False, exclude\_defaults: bool = False, exclude\_none:*

*bool = False, encoder: Optional[Callable[[Any], Any]] = None, models\_as\_dict: bool = True,*

*\*\*dumps\_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

*encoder* is an optional function to supply as *default* to *json.dumps()*, other arguments as per *json.dumps()*.

```

marginal_distribution: Optional[pandas.core.series.Series]
marginal_probabilities() → Optional[List]
marginal_states() → Optional[List]
max() → Any
 Get the max value of the distribution.
min() → Any
 Get the min value of the distribution.
name: str
classmethod parse_file(path: Union[str, pathlib.Path], *, content_type: unicode = None, encoding:
 unicode = 'utf8', proto: pydantic.parse.Protocol = None, allow_pickle: bool =
 False) → Model
classmethod parse_obj(obj: Any) → Model
classmethod parse_raw(b: Union[str, bytes], *, content_type: unicode = None, encoding: unicode = 'utf8',
 proto: pydantic.parse.Protocol = None, allow_pickle: bool = False) → Model
random_state: int
sample(count: int = 1) → Any
 Sample a value from the Distribution.
sample_marginal(count: int = 1) → Any
classmethod schema(by_alias: bool = True, ref_template: unicode = '#/definitions/{model}') →
 DictStrAny
classmethod schema_json(*, by_alias: bool = True, ref_template: unicode = '#/definitions/{model}',
 **dumps_kwargs: Any) → unicode
classmethod update_forward_refs(**localns: Any) → None
 Try to update ForwardRefs on fields based on this Model, globalns and localns.
classmethod validate(value: Any) → Model
class DatetimeDistribution(*, name: str, data: Optional[pandas.core.series.Series] = None, random_state:
 int = 0, marginal_distribution: Optional[pandas.core.series.Series] = None, low:
 datetime.datetime = datetime.datetime(1970, 1, 1, 0, 0), high: datetime.datetime
 = datetime.datetime(2024, 3, 11, 11, 44, 28, 414668), step: datetime.timedelta =
 datetime.timedelta(microseconds=1), offset: datetime.timedelta =
 datetime.timedelta(seconds=120))
Bases: synthcity.plugins.core.distribution.Distribution

```



```

class Config
 Bases: object
 arbitrary_types_allowed = True

```

**as\_constraint()** → *synthcity.plugins.core.constraints.Constraints*

Convert the Distribution to a set of Constraints.

**classmethod construct**(*\_fields\_set: Optional[SetStr] = None, \*\*values: Any*) → Model

Creates a new model setting `__dict__` and `__fields_set__` from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if `Config.extra = 'allow'` was set since it adds all passed values

**copy**(*\*, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, update: Optional[DictStrAny] = None, deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

#### Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to `True` to make a deep copy of the model

**Returns** new model instance

**data:** `Optional[pandas.core.series.Series]`

**dict**(*\*, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, by\_alias: bool = False, skip\_defaults: Optional[bool] = None, exclude\_unset: bool = False, exclude\_defaults: bool = False, exclude\_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

**dtype()** → str

**classmethod from\_orm**(*obj: Any*) → Model

**get()** → List[Any]

Return the metadata of the Distribution.

**has**(*val: datetime.datetime*) → bool

Test if a value is included in the Distribution.

**high:** `datetime.datetime`

**includes**(*other: synthcity.plugins.core.distribution.Distribution*) → bool

Test if another Distribution is included in the local one.

**json**(*\*, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, by\_alias: bool = False, skip\_defaults: Optional[bool] = None, exclude\_unset: bool = False, exclude\_defaults: bool = False, exclude\_none: bool = False, encoder: Optional[Callable[[Any], Any]] = None, models\_as\_dict: bool = True, \*\*kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

*encoder* is an optional function to supply as *default* to *json.dumps()*, other arguments as per *json.dumps()*.

**low:** `datetime.datetime`

**marginal\_distribution:** `Optional[pandas.core.series.Series]`

**marginal\_probabilities()** → Optional[List]

```

marginal_states() → Optional[List]

max() → Any
 Get the max value of the distribution.

min() → Any
 Get the min value of the distribution.

name: str

offset: datetime.timedelta

classmethod parse_file(path: Union[str, pathlib.Path], *, content_type: unicode = None, encoding:
 unicode = 'utf8', proto: pydantic.parse.Protocol = None, allow_pickle: bool =
 False) → Model

classmethod parse_obj(obj: Any) → Model

classmethod parse_raw(b: Union[str, bytes], *, content_type: unicode = None, encoding: unicode = 'utf8',
 proto: pydantic.parse.Protocol = None, allow_pickle: bool = False) → Model

random_state: int

sample(count: int = 1) → Any
 Sample a value from the Distribution.

sample_marginal(count: int = 1) → Any

classmethod schema(by_alias: bool = True, ref_template: unicode = '#/definitions/{model}') →
 DictStrAny

classmethod schema_json(*, by_alias: bool = True, ref_template: unicode = '#/definitions/{model}',
 **dumps_kwargs: Any) → unicode

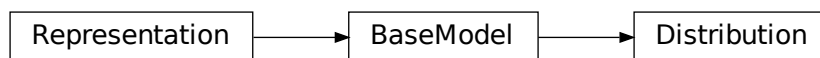
step: datetime.timedelta

classmethod update_forward_refs(**localns: Any) → None
 Try to update ForwardRefs on fields based on this Model, globalns and localns.

classmethod validate(value: Any) → Model

class Distribution(*, name: str, data: Optional[pandas.core.series.Series] = None, random_state: int = 0,
 marginal_distribution: Optional[pandas.core.series.Series] = None)
 Bases: pydantic.main.BaseModel

```



Base class of all Distributions.

The Distribution class characterizes the **empirical** marginal distribution of the feature. Each derived class must implement the following methods:

get() - Return the metadata of the Distribution. sample() - Sample a value from the Distribution.  
includes() - Test if another Distribution is included in the local one. has() - Test if a value is included  
in the support of the Distribution. as\_constraint() - Convert the Distribution to a set of Constraints.

`min()` - Return the minimum of the support. `max()` - Return the maximum of the support. `__eq__()`  
- Testing equality of two Distributions. `dtype()` - Return the data type

Examples of derived classes include `CategoricalDistribution`, `FloatDistribution`, and `IntegerDistribution`.

**class** `Config`

Bases: `object`

**arbitrary\_types\_allowed** = `True`

**abstract** `as_constraint()` → [synthcity.plugins.core.constraints.Constraints](#)

Convert the Distribution to a set of Constraints.

**classmethod** `construct(_fields_set: Optional[SetStr] = None, **values: Any) → Model`

Creates a new model setting `__dict__` and `__fields_set__` from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if `Config.extra = 'allow'` was set since it adds all passed values

**copy**(`*`, `include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude:`

`Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, update: Optional[DictStrAny] = None, deep: bool = False`) → `Model`

Duplicate a model, optionally choose which fields to include, exclude and change.

#### Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to `True` to make a deep copy of the model

**Returns** new model instance

**data:** `Optional[pandas.core.series.Series]`

**dict**(`*`, `include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude:`

`Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, by_alias: bool = False, skip_defaults: Optional[bool] = None, exclude_unset: bool = False, exclude_defaults: bool = False, exclude_none: bool = False`) → `DictStrAny`

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

**abstract** `dtype()` → `str`

**classmethod** `from_obj(obj: Any) → Model`

**abstract** `get()` → `List[Any]`

Return the metadata of the Distribution.

**abstract** `has(val: Any) → bool`

Test if a value is included in the Distribution.

**abstract** `includes(other: synthcity.plugins.core.distribution.Distribution) → bool`

Test if another Distribution is included in the local one.

**json**(`*`, `include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude:`

`Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, by_alias: bool = False, skip_defaults: Optional[bool] = None, exclude_unset: bool = False, exclude_defaults: bool = False, exclude_none: bool = False, encoder: Optional[Callable[[Any], Any]] = None, models_as_dict: bool = True, **dumps_kwargs: Any`) → `unicode`

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.



*encoder* is an optional function to supply as *default* to `json.dumps()`, other arguments as per `json.dumps()`.

```

marginal_distribution: Optional[pandas.core.series.Series]
marginal_probabilities() → Optional[List]
marginal_states() → Optional[List]
abstract max() → Any
 Get the max value of the distribution.
abstract min() → Any
 Get the min value of the distribution.
name: str
classmethod parse_file(path: Union[str, pathlib.Path], *, content_type: unicode = None, encoding:
 unicode = 'utf8', proto: pydantic.parse.Protocol = None, allow_pickle: bool =
 False) → Model
classmethod parse_obj(obj: Any) → Model
classmethod parse_raw(b: Union[str, bytes], *, content_type: unicode = None, encoding: unicode = 'utf8',
 proto: pydantic.parse.Protocol = None, allow_pickle: bool = False) → Model
random_state: int
abstract sample(count: int = 1) → Any
 Sample a value from the Distribution.
sample_marginal(count: int = 1) → Any
classmethod schema(by_alias: bool = True, ref_template: unicode = '#/definitions/{model}') →
 DictStrAny
classmethod schema_json(*, by_alias: bool = True, ref_template: unicode = '#/definitions/{model}',
 **dumps_kwargs: Any) → unicode
classmethod update_forward_refs(**localns: Any) → None
 Try to update ForwardRefs on fields based on this Model, globalns and localns.
classmethod validate(value: Any) → Model
class FloatDistribution(*, name: str, data: Optional[pandas.core.series.Series] = None, random_state: int =
 0, marginal_distribution: Optional[pandas.core.series.Series] = None, low: float = -
 1.7976931348623157e+308, high: float = 1.7976931348623157e+308)
 Bases: synthcity.plugins.core.distribution.Distribution

```



```

class Config
 Bases: object
 arbitrary_types_allowed = True
as_constraint() → synthcity.plugins.core.constraints.Constraints
 Convert the Distribution to a set of Constraints.

```

**classmethod** **construct**(*\_fields\_set: Optional[SetStr] = None, \*\*values: Any*) → Model

Creates a new model setting `__dict__` and `__fields_set__` from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if `Config.extra = 'allow'` was set since it adds all passed values

**copy**(*\*, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, update: Optional[DictStrAny] = None, deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

#### Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to `True` to make a deep copy of the model

**Returns** new model instance

**data:** `Optional[pandas.core.series.Series]`

**dict**(*\*, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, by\_alias: bool = False, skip\_defaults: Optional[bool] = None, exclude\_unset: bool = False, exclude\_defaults: bool = False, exclude\_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

**dtype**() → str

**classmethod** **from\_orm**(*obj: Any*) → Model

**get**() → List[Any]

Return the metadata of the Distribution.

**has**(*val: Any*) → bool

Test if a value is included in the Distribution.

**high:** float

**includes**(*other: synthcity.plugins.core.distribution.Distribution*) → bool

Test if another Distribution is included in the local one.

**json**(*\*, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, by\_alias: bool = False, skip\_defaults: Optional[bool] = None, exclude\_unset: bool = False, exclude\_defaults: bool = False, exclude\_none: bool = False, encoder: Optional[Callable[[Any], Any]] = None, models\_as\_dict: bool = True, \*\*dumps\_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict*().

*encoder* is an optional function to supply as *default* to *json.dumps()*, other arguments as per *json.dumps()*.

**low:** float

**marginal\_distribution:** `Optional[pandas.core.series.Series]`

**marginal\_probabilities**() → Optional[List]

**marginal\_states**() → Optional[List]

```

max() → Any
 Get the max value of the distribution.

min() → Any
 Get the min value of the distribution.

name: str

classmethod parse_file(path: Union[str, pathlib.Path], *, content_type: unicode = None, encoding: unicode = 'utf8', proto: pydantic.parse.Protocol = None, allow_pickle: bool = False) → Model

classmethod parse_obj(obj: Any) → Model

classmethod parse_raw(b: Union[str, bytes], *, content_type: unicode = None, encoding: unicode = 'utf8', proto: pydantic.parse.Protocol = None, allow_pickle: bool = False) → Model

random_state: int

sample(count: int = 1) → Any
 Sample a value from the Distribution.

sample_marginal(count: int = 1) → Any

classmethod schema(by_alias: bool = True, ref_template: unicode = '#/definitions/{model}') → DictStrAny

classmethod schema_json(*, by_alias: bool = True, ref_template: unicode = '#/definitions/{model}', **kwargs: Any) → unicode

classmethod update_forward_refs(**localns: Any) → None
 Try to update ForwardRefs on fields based on this Model, globalns and localns.

classmethod validate(value: Any) → Model

class IntLogDistribution(*, name: str, data: Optional[pandas.core.series.Series] = None, random_state: int = 0, marginal_distribution: Optional[pandas.core.series.Series] = None, low: int = 1, high: int = 9223372036854775807, step: int = 1)
 Bases: synthcity.plugins.core.distribution.IntegerDistribution

class Config
 Bases: object

 arbitrary_types_allowed = True

as_constraint() → synthcity.plugins.core.constraints.Constraints
 Convert the Distribution to a set of Constraints.

classmethod construct(_fields_set: Optional[SetStr] = None, **values: Any) → Model
 Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if Config.extra = 'allow' was set since it adds all passed values

copy(*, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, update: Optional[DictStrAny] = None, deep: bool = False) → Model
 Duplicate a model, optionally choose which fields to include, exclude and change.

```

#### Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include

- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

**Returns** new model instance

**data:** `Optional[pandas.core.series.Series]`

**dict**(\*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, by\_alias: bool = False, skip\_defaults: Optional[bool] = None, exclude\_unset: bool = False, exclude\_defaults: bool = False, exclude\_none: bool = False*) → `DictStrAny`  
Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

**dtype()** → `str`

**classmethod from\_orm**(*obj: Any*) → `Model`

**get()** → `List[Any]`  
Return the metadata of the Distribution.

**has**(*val: Any*) → `bool`  
Test if a value is included in the Distribution.

**high:** `int`

**includes**(*other: synthcity.plugins.core.distribution.Distribution*) → `bool`  
Test if another Distribution is included in the local one.

**json**(\*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, by\_alias: bool = False, skip\_defaults: Optional[bool] = None, exclude\_unset: bool = False, exclude\_defaults: bool = False, exclude\_none: bool = False, encoder: Optional[Callable[[Any], Any]] = None, models\_as\_dict: bool = True, \*\*dumps\_kwargs: Any*) → `unicode`  
Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.  
*encoder* is an optional function to supply as *default* to *json.dumps()*, other arguments as per *json.dumps()*.

**low:** `int`

**marginal\_distribution:** `Optional[pandas.core.series.Series]`

**marginal\_probabilities()** → `Optional[List]`

**marginal\_states()** → `Optional[List]`

**max()** → `Any`  
Get the max value of the distribution.

**min()** → `Any`  
Get the min value of the distribution.

**name:** `str`

**classmethod parse\_file**(*path: Union[str, pathlib.Path], \*, content\_type: unicode = None, encoding: unicode = 'utf8', proto: pydantic.parse.Protocol = None, allow\_pickle: bool = False*) → `Model`

**classmethod parse\_obj**(*obj: Any*) → `Model`

**classmethod parse\_raw**(*b: Union[str, bytes], \*, content\_type: unicode = None, encoding: unicode = 'utf8', proto: pydantic.parse.Protocol = None, allow\_pickle: bool = False*) → `Model`

**random\_state:** `int`

```

sample(count: int = 1) → Any
 Sample a value from the Distribution.

sample_marginal(count: int = 1) → Any

classmethod schema(by_alias: bool = True, ref_template: unicode = '#/definitions/{model}') →
 DictStrAny

classmethod schema_json(*, by_alias: bool = True, ref_template: unicode = '#/definitions/{model}',
 **dumps_kwargs: Any) → unicode

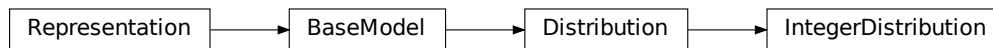
step: int

classmethod update_forward_refs(**localns: Any) → None
 Try to update ForwardRefs on fields based on this Model, globalns and localns.

classmethod validate(value: Any) → Model

class IntegerDistribution(*, name: str, data: Optional[pandas.core.series.Series] = None, random_state: int
 = 0, marginal_distribution: Optional[pandas.core.series.Series] = None, low: int
 = - 9223372036854775808, high: int = 9223372036854775807, step: int = 1)
 Bases: synthcity.plugins.core.distribution.Distribution

```



```

class Config
 Bases: object

 arbitrary_types_allowed = True

as_constraint() → synthcity.plugins.core.constraints.Constraints
 Convert the Distribution to a set of Constraints.

classmethod construct(_fields_set: Optional[SetStr] = None, **values: Any) → Model
 Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values
 are respected, but no other validation is performed. Behaves as if Config.extra = 'allow' was set since it
 adds all passed values

copy(*, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude:
 Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, update: Optional[DictStrAny] = None,
 deep: bool = False) → Model
 Duplicate a model, optionally choose which fields to include, exclude and change.

```

#### Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

**Returns** new model instance

**data:** `Optional[pandas.core.series.Series]`

**dict**(\**include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, by\_alias: bool = False, skip\_defaults: Optional[bool] = None, exclude\_unset: bool = False, exclude\_defaults: bool = False, exclude\_none: bool = False*) → `DictStrAny`  
Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

**dtype**() → `str`

**classmethod from\_orm**(*obj: Any*) → `Model`

**get**() → `List[Any]`  
Return the metadata of the Distribution.

**has**(*val: Any*) → `bool`  
Test if a value is included in the Distribution.

**high:** `int`

**includes**(*other: synthcity.plugins.core.distribution.Distribution*) → `bool`  
Test if another Distribution is included in the local one.

**json**(\**include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, by\_alias: bool = False, skip\_defaults: Optional[bool] = None, exclude\_unset: bool = False, exclude\_defaults: bool = False, exclude\_none: bool = False, encoder: Optional[Callable[[Any], Any]] = None, models\_as\_dict: bool = True, \*\*dumps\_kwargs: Any*) → `unicode`  
Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict*().  
*encoder* is an optional function to supply as *default* to *json.dumps()*, other arguments as per *json.dumps()*.

**low:** `int`

**marginal\_distribution:** `Optional[pandas.core.series.Series]`

**marginal\_probabilities**() → `Optional[List]`

**marginal\_states**() → `Optional[List]`

**max**() → `Any`  
Get the max value of the distribution.

**min**() → `Any`  
Get the min value of the distribution.

**name:** `str`

**classmethod parse\_file**(*path: Union[str, pathlib.Path], \*, content\_type: unicode = None, encoding: unicode = 'utf8', proto: pydantic.parse.Protocol = None, allow\_pickle: bool = False*) → `Model`

**classmethod parse\_obj**(*obj: Any*) → `Model`

**classmethod parse\_raw**(*b: Union[str, bytes], \*, content\_type: unicode = None, encoding: unicode = 'utf8', proto: pydantic.parse.Protocol = None, allow\_pickle: bool = False*) → `Model`

**random\_state:** `int`

**sample**(*count: int = 1*) → `Any`  
Sample a value from the Distribution.

**sample\_marginal**(*count: int = 1*) → `Any`

```

classmethod schema(by_alias: bool = True, ref_template: unicode = '#/definitions/{model}') → DictStrAny

classmethod schema_json(* , by_alias: bool = True, ref_template: unicode = '#/definitions/{model}',
 **dumps_kwargs: Any) → unicode

step: int

classmethod update_forward_refs(**localns: Any) → None
 Try to update ForwardRefs on fields based on this Model, globalns and localns.

classmethod validate(value: Any) → Model

class LogDistribution(* , name: str, data: Optional[pandas.core.series.Series] = None, random_state: int = 0,
 marginal_distribution: Optional[pandas.core.series.Series] = None, low: float =
 2.2250738585072014e-308, high: float = 1.7976931348623157e+308)

Bases: synthcity.plugins.core.distribution.FloatDistribution

class Config
 Bases: object

 arbitrary_types_allowed = True

as_constraint() → synthcity.plugins.core.constraints.Constraints
 Convert the Distribution to a set of Constraints.

classmethod construct(_fields_set: Optional[SetStr] = None, **values: Any) → Model
 Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values
 are respected, but no other validation is performed. Behaves as if Config.extra = 'allow' was set since it
 adds all passed values

copy(* , include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude:
 Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, update: Optional[DictStrAny] = None,
 deep: bool = False) → Model
 Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

 • include – fields to include in new model

 • exclude – fields to exclude from new model, as with values this takes precedence
 over include

 • update – values to change/add in the new model. Note: the data is not validated before
 creating the new model: you should trust this data

 • deep – set to True to make a deep copy of the model

Returns new model instance

data: Optional[pandas.core.series.Series]

dict(* , include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude:
 Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, by_alias: bool = False, skip_defaults:
 Optional[bool] = None, exclude_unset: bool = False, exclude_defaults: bool = False, exclude_none:
 bool = False) → DictStrAny
 Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

dtype() → str

classmethod from_orm(obj: Any) → Model

get() → List[Any]
 Return the metadata of the Distribution.

```

**has**(*val: Any*) → bool

Test if a value is included in the Distribution.

**high**: float

**includes**(*other: synthcity.plugins.core.distribution.Distribution*) → bool

Test if another Distribution is included in the local one.

**json**(*\*, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, by\_alias: bool = False, skip\_defaults: Optional[bool] = None, exclude\_unset: bool = False, exclude\_defaults: bool = False, exclude\_none: bool = False, encoder: Optional[Callable[[Any], Any]] = None, models\_as\_dict: bool = True, \*\*dumps\_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

*encoder* is an optional function to supply as *default* to *json.dumps()*, other arguments as per *json.dumps()*.

**low**: float

**marginal\_distribution**: Optional[pandas.core.series.Series]

**marginal\_probabilities**() → Optional[List]

**marginal\_states**() → Optional[List]

**max**() → Any

Get the max value of the distribution.

**min**() → Any

Get the min value of the distribution.

**name**: str

**classmethod parse\_file**(*path: Union[str, pathlib.Path], \*, content\_type: unicode = None, encoding: unicode = 'utf8', proto: pydantic.parse.Protocol = None, allow\_pickle: bool = False*) → Model

**classmethod parse\_obj**(*obj: Any*) → Model

**classmethod parse\_raw**(*b: Union[str, bytes], \*, content\_type: unicode = None, encoding: unicode = 'utf8', proto: pydantic.parse.Protocol = None, allow\_pickle: bool = False*) → Model

**random\_state**: int

**sample**(*count: int = 1*) → Any

Sample a value from the Distribution.

**sample\_marginal**(*count: int = 1*) → Any

**classmethod schema**(*by\_alias: bool = True, ref\_template: unicode = '#/definitions/{model}'*) → DictStrAny

**classmethod schema\_json**(*\*, by\_alias: bool = True, ref\_template: unicode = '#/definitions/{model}', \*\*dumps\_kwargs: Any*) → unicode

**classmethod update\_forward\_refs**(*\*\*localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

**classmethod validate**(*value: Any*) → Model

**constraint\_to\_distribution**(*constraints: synthcity.plugins.core.constraints.Constraints, feature: str*) → *synthcity.plugins.core.distribution.Distribution*

Infer Distribution from Constraints.

### Parameters

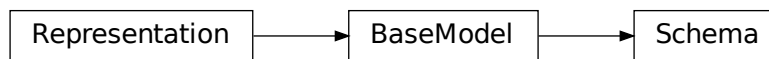


- **constraints** – Constraints The Constraints on features.
- **feature** – str The name of the feature in question.

**Returns** The inferred Distribution.

## synthcity.plugins.core.schema module

```
class Schema(*, sampling_strategy: str = 'marginal', protected_cols: List[str] = ['seq_id'], random_state: int = 0,
 data: Any = None, domain: Dict = {})
 Bases: pydantic.main.BaseModel
```



Utility class for defining the schema of a Dataset.

### Constructor Args:

**domain: Dict** A dictionary of feature\_name: Distribution.

**sampling\_strategy: str** Taking value of “marginal” (default) or “uniform” (for debugging).

**protected\_cols: List[str]** List of columns that are exempt from distributional constraints (e.g. ID column)

**random\_state: int** Random seed (default 0)

**data: Any** (Optional) the data set

### Config

alias of `pydantic.config.BaseConfig`

**adapt\_dtypes**(X: *pandas.core.frame.DataFrame*) → *pandas.core.frame.DataFrame*

Applying the data type to a new data frame

**Parameters** X – *pd.DataFrame* A new data frame to be adapted.

**Returns** A data frame whose data types are coerced to be the same with the Schema. If the data frame contains new features, these will be retained as is.

**as\_constraints**() → *synthcity.plugins.core.constraints.Constraints*

Convert the schema to a list of Constraints.

**classmethod construct**(*\_fields\_set: Optional[SetStr] = None, \*\*values: Any*) → *Model*

Creates a new model setting `__dict__` and `__fields_set__` from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if `Config.extra = 'allow'` was set since it adds all passed values

**copy**(\*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, update: Optional[DictStrAny] = None, deep: bool = False*) → *Model*

Duplicate a model, optionally choose which fields to include, exclude and change.

### Parameters

- **include** – fields to include in new model

- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

**Returns** new model instance

**data:** Any

**dict**(\*, include: *Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, exclude: *Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, by\_alias: *bool = False*, skip\_defaults: *Optional[bool] = None*, exclude\_unset: *bool = False*, exclude\_defaults: *bool = False*, exclude\_none: *bool = False*) → DictStrAny  
Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

**domain:** Dict

**features()** → List

**classmethod from\_constraints**(constraints: *synthcity.plugins.core.constraints.Constraints*) → *synthcity.plugins.core.schema.Schema*  
Create a schema from a list of Constraints.

**classmethod from\_orm**(obj: Any) → Model

**get**(feature: str) → *synthcity.plugins.core.distribution.Distribution*  
Get the Distribution of a feature.

**Parameters** **feature** – str. the feature name

**Returns** The feature distribution

**includes**(other: *synthcity.plugins.core.schema.Schema*) → bool  
Test if another schema is included in the local one.

**json**(\*, include: *Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, exclude: *Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, by\_alias: *bool = False*, skip\_defaults: *Optional[bool] = None*, exclude\_unset: *bool = False*, exclude\_defaults: *bool = False*, exclude\_none: *bool = False*, encoder: *Optional[Callable[[Any], Any]] = None*, models\_as\_dict: *bool = True*, \*\*dumps\_kwargs: Any) → unicode  
Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.  
*encoder* is an optional function to supply as *default* to *json.dumps()*, other arguments as per *json.dumps()*.

**classmethod parse\_file**(path: *Union[str, pathlib.Path]*, \*, content\_type: *unicode = None*, encoding: *unicode = 'utf8'*, proto: *pydantic.parse.Protocol = None*, allow\_pickle: *bool = False*) → Model

**classmethod parse\_obj**(obj: Any) → Model

**classmethod parse\_raw**(b: *Union[str, bytes]*, \*, content\_type: *unicode = None*, encoding: *unicode = 'utf8'*, proto: *pydantic.parse.Protocol = None*, allow\_pickle: *bool = False*) → Model

**protected\_cols:** List[str]

**random\_state:** int

**sample**(count: int) → pandas.core.frame.DataFrame

**sampling\_strategy:** str

```

classmethod schema(by_alias: bool = True, ref_template: unicode = '#/definitions/{model}') → DictStrAny
classmethod schema_json(* , by_alias: bool = True, ref_template: unicode = '#/definitions/{model}', **dumps_kwargs: Any) → unicode
classmethod update_forward_refs(**localns: Any) → None
 Try to update ForwardRefs on fields based on this Model, globalns and localns.
classmethod validate(value: Any) → Model

```

## synthcity.plugins.core.serializable module

```

class Serializable(*args: Any, **kwargs: Any)
 Bases: object
 Utility class for model persistence.
 static load(buff: bytes) → Any
 static load_dict(representation: dict) → Any
 save() → bytes
 save_dict() → dict
 save_to_file(path: pathlib.Path) → bytes
 static version() → str
 API version

```

## 12.1.2 Core model implementations

### synthcity.plugins.core.models.tabular\_encoder module

TabularEncoder module.

```

class BinEncoder(*args: Any, **kwargs: Any)
 Bases: synthcity.plugins.core.models.tabular_encoder.TabularEncoder
 Binary encoder (for SurvivalGAN).
 Model continuous columns with a BayesianGMM and normalized to a scalar [0, 1] and a vector. Discrete columns
 are encoded using a scikit-learn OneHotEncoder.
 activation_layout(discrete_activation: str, continuous_activation: str) → Sequence[Tuple[str, int]]
 Get the layout of the activations.
 Returns a list of tuple, describing each column as:
 • continuous, and with length 1 + number of GMM clusters.
 • discrete, and with length <N>, the length of the one-hot encoding.
 cat_encoder_params: dict = {}
 categorical_encoder: Union[str, type] = 'passthrough'
 cont_encoder_params: dict = {'n_components': 2}
 continuous_encoder: Union[str, type] = 'bayesian_gmm'

```

**fit**(raw\_data: pandas.core.frame.DataFrame, discrete\_columns: Optional[List] = None) → Any  
Fit the TabularEncoder.

This step also counts the #columns in matrix data and span information.

**get\_column\_info**(name: str) → synthcity.plugins.core.models.tabular\_encoder.FeatureInfo

**inverse\_transform**(data: pandas.core.frame.DataFrame) → pandas.core.frame.DataFrame  
Take matrix data and output raw data.

Output uses the same type as input to the transform function.

**layout**() → Sequence[synthcity.plugins.core.models.tabular\_encoder.FeatureInfo]  
Get the layout of the encoded dataset.

**Returns a list of tuple, describing each column as:**

- continuous, and with length 1 + number of GMM clusters.
- discrete, and with length <N>, the length of the one-hot encoding.

**n\_features**() → int

**transform**(raw\_data: pandas.core.frame.DataFrame) → pandas.core.frame.DataFrame  
Take raw data and output a matrix data.

**class FeatureInfo**(\*, name: str, feature\_type: str, transform: Any = None, output\_dimensions: int, transformed\_features: List[str], trans\_feature\_types: List[str])  
Bases: pydantic.main.BaseModel

**Config**

alias of pydantic.config.BaseConfig

**classmethod construct**(\_fields\_set: Optional[SetStr] = None, \*\*values: Any) → Model  
Creates a new model setting \_\_dict\_\_ and \_\_fields\_set\_\_ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if Config.extra = 'allow' was set since it adds all passed values

**copy**(\*, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, update: Optional[DictStrAny] = None, deep: bool = False) → Model  
Duplicate a model, optionally choose which fields to include, exclude and change.

**Parameters**

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to True to make a deep copy of the model

**Returns** new model instance

**dict**(\*, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, by\_alias: bool = False, skip\_defaults: Optional[bool] = None, exclude\_unset: bool = False, exclude\_defaults: bool = False, exclude\_none: bool = False) → DictStrAny  
Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

**feature\_type**: str

```

classmethod from_orm(obj: Any) → Model

json(*, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude:
 Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, by_alias: bool = False, skip_defaults:
 Optional[bool] = None, exclude_unset: bool = False, exclude_defaults: bool = False, exclude_none:
 bool = False, encoder: Optional[Callable[[Any], Any]] = None, models_as_dict: bool = True,
 **kwargs: Any) → unicode
 Generate a JSON representation of the model, include and exclude arguments as per dict().

 encoder is an optional function to supply as default to json.dumps(), other arguments as per json.dumps().

name: str
output_dimensions: int

classmethod parse_file(path: Union[str, pathlib.Path], *, content_type: unicode = None, encoding:
 unicode = 'utf8', proto: pydantic.parse.Protocol = None, allow_pickle: bool =
 False) → Model

classmethod parse_obj(obj: Any) → Model

classmethod parse_raw(b: Union[str, bytes], *, content_type: unicode = None, encoding: unicode = 'utf8',
 proto: pydantic.parse.Protocol = None, allow_pickle: bool = False) → Model

classmethod schema(by_alias: bool = True, ref_template: unicode = '#/definitions/{model}') →
 DictStrAny

classmethod schema_json(*, by_alias: bool = True, ref_template: unicode = '#/definitions/{model}',
 **kwargs: Any) → unicode

trans_feature_types: List[str]
transform: Any
transformed_features: List[str]

classmethod update_forward_refs(**localns: Any) → None
 Try to update ForwardRefs on fields based on this Model, globalns and localns.

classmethod validate(value: Any) → Model

class TabularEncoder(*args: Any, **kwargs: Any)
 Bases: sklearn.base.TransformerMixin, sklearn.base.BaseEstimator

 Tabular encoder.

 Model continuous columns with a BayesianGMM and normalized to a scalar [0, 1] and a vector. Discrete columns
 are encoded using a scikit-learn OneHotEncoder.

activation_layout(discrete_activation: str, continuous_activation: str) → Sequence[Tuple[str, int]]
 Get the layout of the activations.

 Returns a list of tuple, describing each column as:
 • continuous, and with length 1 + number of GMM clusters.
 • discrete, and with length <N>, the length of the one-hot encoding.

cat_encoder_params: dict = {'handle_unknown': 'ignore', 'sparse_output': False}
categorical_encoder: Union[str, type] = 'onehot'
cont_encoder_params: dict = {'n_components': 10}
continuous_encoder: Union[str, type] = 'bayesian_gmm'

```

**fit**(*raw\_data*: *pandas.core.frame.DataFrame*, *discrete\_columns*: *Optional[List] = None*) → Any  
Fit the TabularEncoder.

This step also counts the #columns in matrix data and span information.

**get\_column\_info**(*name*: *str*) → *synthcity.plugins.core.models.tabular\_encoder.FeatureInfo*

**inverse\_transform**(*data*: *pandas.core.frame.DataFrame*) → *pandas.core.frame.DataFrame*  
Take matrix data and output raw data.

Output uses the same type as input to the transform function.

**layout**() → *Sequence[synthcity.plugins.core.models.tabular\_encoder.FeatureInfo]*  
Get the layout of the encoded dataset.

**Returns a list of tuple, describing each column as:**

- continuous, and with length 1 + number of GMM clusters.
- discrete, and with length <N>, the length of the one-hot encoding.

**n\_features**() → int

**transform**(*raw\_data*: *pandas.core.frame.DataFrame*) → *pandas.core.frame.DataFrame*  
Take raw data and output a matrix data.

**class TimeSeriesBinEncoder**(\*args: Any, \*\*kwargs: Any)  
Bases: *sklearn.base.TransformerMixin*, *sklearn.base.BaseEstimator*  
Time series Bin encoder.

Model continuous columns with a BayesianGMM and normalized to a scalar [0, 1] and a vector. Discrete columns are encoded using a scikit-learn OneHotEncoder.

**fit**(*static\_data*: *pandas.core.frame.DataFrame*, *temporal\_data*: *List[pandas.core.frame.DataFrame]*,  
*observation\_times*: *List*, *discrete\_columns*: *Optional[List] = None*) →  
*synthcity.plugins.core.models.tabular\_encoder.TimeSeriesBinEncoder*  
Fit the TimeSeriesBinEncoder

**fit\_transform**(*static*: *pandas.core.frame.DataFrame*, *temporal*: *List[pandas.core.frame.DataFrame]*,  
*observation\_times*: *List*) → *pandas.core.frame.DataFrame*

**transform**(*static\_data*: *pandas.core.frame.DataFrame*, *temporal\_data*: *List[pandas.core.frame.DataFrame]*,  
*observation\_times*: *List*) → *pandas.core.frame.DataFrame*  
Take raw data and output a matrix data.

**class TimeSeriesTabularEncoder**(\*args: Any, \*\*kwargs: Any)  
Bases: *sklearn.base.TransformerMixin*, *sklearn.base.BaseEstimator*  
TimeSeries Tabular encoder.

Model continuous columns with a BayesianGMM and normalized to a scalar [0, 1] and a vector. Discrete columns are encoded using a scikit-learn OneHotEncoder.

**activation\_layout**(*discrete\_activation*: *str*, *continuous\_activation*: *str*) → *Tuple*

**activation\_layout\_temporal**(*discrete\_activation*: *str*, *continuous\_activation*: *str*) → Any

**fit**(*static\_data*: *pandas.core.frame.DataFrame*, *temporal\_data*: *List[pandas.core.frame.DataFrame]*,  
*observation\_times*: *List*, *discrete\_columns*: *Optional[List] = None*) →  
*synthcity.plugins.core.models.tabular\_encoder.TimeSeriesTabularEncoder*

**fit\_temporal**(*temporal\_data*: *List[pandas.core.frame.DataFrame]*, *observation\_times*: *List*,  
*discrete\_columns*: *Optional[List] = None*) →  
*synthcity.plugins.core.models.tabular\_encoder.TimeSeriesTabularEncoder*

```

fit_transform(static_data: pandas.core.frame.DataFrame, temporal_data:
 List[pandas.core.frame.DataFrame], observation_times: List) →
 Tuple[pandas.core.frame.DataFrame, pandas.core.frame.DataFrame, List]

fit_transform_temporal(temporal_data: List[pandas.core.frame.DataFrame], observation_times: List)
 → Tuple[pandas.core.frame.DataFrame, List]

inverse_transform(static_encoded: pandas.core.frame.DataFrame, temporal_encoded:
 List[pandas.core.frame.DataFrame], observation_times: List) →
 pandas.core.frame.DataFrame

inverse_transform_observation_times(observation_times: List) → pandas.core.frame.DataFrame

inverse_transform_static(static_encoded: pandas.core.frame.DataFrame) →
 pandas.core.frame.DataFrame

inverse_transform_temporal(temporal_encoded: List[pandas.core.frame.DataFrame],
 observation_times: List) → pandas.core.frame.DataFrame

layout() → Tuple[List, List]

n_features() → Tuple

transform(static_data: pandas.core.frame.DataFrame, temporal_data: List[pandas.core.frame.DataFrame],
 observation_times: List) → Tuple[pandas.core.frame.DataFrame, pandas.core.frame.DataFrame,
 List]

transform_observation_times(observation_times: List) → List

transform_static(static_data: pandas.core.frame.DataFrame) → pandas.core.frame.DataFrame

transform_temporal(temporal_data: List[pandas.core.frame.DataFrame], observation_times: List) →
 Tuple[pandas.core.frame.DataFrame, List]

```

## synthcity.plugins.core.models.mlp module

```

class LinearLayer(n_units_in: int, n_units_out: int, dropout: float = 0, batch_norm: bool = False, nonlin:
 Optional[str] = 'relu', device: Any = device(type='cpu'))

```

Bases: torch.nn.modules.module.Module

### T\_destination

alias of T\_destination, bound=Dict[str, Any])

```

add_module(name: str, module: Optional[torch.nn.modules.module.Module]) → None

```

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

### Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

```

apply(fn: Callable[[torch.nn.modules.module.Module], None]) → torch.nn.modules.module.T

```

Applies fn recursively to every submodule (as returned by .children()) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

**Parameters** **fn** (*Module* → None) – function to be applied to each submodule

**Returns** self

**Return type** Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**buffers**(recurse: bool = True) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (bool) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** torch.Tensor – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** Module – a child module

**cpu()** → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.



---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**double**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to double datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**dump\_patches: bool = False**

**eval**() → torch.nn.modules.module.T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** self

**Return type** Module

**extra\_repr**() → str

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**float**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to float datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**forward**(*X: torch.Tensor*) → `torch.Tensor`

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**get\_buffer**(*target: str*) → `torch.Tensor`

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

**Parameters** `target` – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by `target`

**Return type** `torch.Tensor`

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → `Any`

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** `object`

**get\_parameter**(*target: str*) → `torch.nn.parameter.Parameter`

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

**Parameters** `target` – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by `target`

**Return type** `torch.nn.Parameter`

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by **target** if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)
```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** **target** – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by **target**

**Return type** torch.nn.Module

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**half**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** Module

**ipu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** Module

**load\_state\_dict**(*state\_dict*: Mapping[str, Any], *strict*: bool = True)

Copies parameters and buffers from *state\_dict* into this module and its descendants. If *strict* is True, then the keys of *state\_dict* must exactly match the keys returned by this module's `state_dict()` function.

**Parameters**

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in *state\_dict* match the keys returned by this module's `state_dict()` function. Default: True

**Returns**

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** NamedTuple with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as `None` and its corresponding key exists in *state\_dict*, `load_state_dict()` will raise a `RuntimeError`.

---

**modules**() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

**Parameters**

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str*, *torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children()** → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str*, *Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo*: Optional[Set[torch.nn.modules.module.Module]] = None, *prefix*: str = "", *remove\_duplicate*: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str*, *Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str*, *Parameter*) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → `Iterator[torch.nn.parameter.Parameter]`

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → `torch.utils.hooks.RemovableHandle`

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_buffer**(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → `None`

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm’s `running_mean` is not a parameter, but is part of the module’s state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module’s `state_dict`.

Buffers can be accessed as attributes using given names.

**Parameters**

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as `cuda`, are ignored. If `None`, the buffer is **not** included in the module’s `state_dict`.

- **persistent** (*bool*) – whether the buffer is part of this module’s *state\_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → *torch.utils.hooks.RemovableHandle*

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The *grad\_input* and *grad\_output* are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of *grad\_input* in subsequent computations. *grad\_input* will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in *grad\_input* and *grad\_output* will be *None* for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module’s forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Alias for `add_module()`.

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad\_**(*requires\_grad: bool = True*) → `torch.nn.modules.module.T`

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** **requires\_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** `self`

**Return type** `Module`



**set\_extra\_state**(*state: Any*)

This function is called from `load_state_dict()` to handle any extra state found within the *state\_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state\_dict*.

**Parameters** *state (dict)* – Extra state from the *state\_dict*

**share\_memory**() → torch.nn.modules.module.T

See `torch.Tensor.share_memory_()`

**state\_dict**(\*args, destination=None, prefix="", keep\_vars=False)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently `state_dict()` also accepts positional arguments for *destination*, *prefix* and *keep\_vars* in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument *destination* as it is not designed for end-users.

### Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in *state\_dict*. Default: `''`.
- **keep\_vars** (*bool, optional*) – by default the `Tensor`s returned in the *state dict* are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(\*args, \*\*kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(*device=None, dtype=None, non\_blocking=False*)

**to**(*dtype, non\_blocking=False*)

```
to(tensor, non_blocking=False)
```

```
to(memory_format=torch.channels_last)
```

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** `self`

**Return type** `Module`

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```

Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (type or string) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (*int*, optional) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

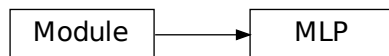
**zero\_grad**(*set\_to\_none: bool = False*) → `None`

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** `set_to_none` (*bool*) – instead of setting to zero, set the grads to `None`. See `torch.optim.Optimizer.zero_grad()` for details.

**class MLP**(*n\_units\_in: int, n\_units\_out: int, \*, task\_type: str = 'regression', n\_layers\_hidden: int = 1, n\_units\_hidden: int = 100, nonlin: str = 'relu', nonlin\_out: Optional[List[Tuple[str, int]]] = None, lr: float = 0.001, weight\_decay: float = 0.001, opt\_betas: tuple = (0.9, 0.999), n\_iter: int = 1000, batch\_size: int = 500, n\_iter\_print: int = 100, random\_state: int = 0, patience: int = 10, n\_iter\_min: int = 100, dropout: float = 0.1, clipping\_value: int = 1, batch\_norm: bool = False, early\_stopping: bool = True, residual: bool = False, loss: Optional[Callable] = None, device: Any = device(type='cpu')*)

Bases: `torch.nn.modules.module.Module`



Fully connected or residual neural nets for classification and regression.

#### Parameters

- **task\_type** (*str*) – classification or regression
- **n\_units\_int** (*int*) – Number of features
- **n\_units\_out** (*int*) – Number of outputs
- **n\_layers\_hidden** (*int*) – Number of hidden layers
- **n\_units\_hidden** (*int*) – Number of hidden units in each layer
- **nonlin** (*string*, default 'elu') – Nonlinearity to use in NN. Can be 'elu', 'relu', 'selu', 'tanh' or 'leaky\_relu'.
- **lr** (*float*) – learning rate for optimizer.
- **weight\_decay** (*float*) – l2 (ridge) penalty for the weights.
- **n\_iter** (*int*) – Maximum number of iterations.
- **batch\_size** (*int*) – Batch size
- **n\_iter\_print** (*int*) – Number of iterations after which to print updates and check the validation loss.

- **random\_state** (*int*) – random\_state used
- **patience** (*int*) – Number of iterations to wait before early stopping after decrease in validation loss
- **n\_iter\_min** (*int*) – Minimum number of iterations to go through before starting early stopping
- **dropout** (*float*) – Dropout value. If 0, the dropout is not used.
- **clipping\_value** (*int*, *default 1*) – Gradients clipping value
- **batch\_norm** (*bool*) – Enable/disable batch norm
- **early\_stopping** (*bool*) – Enable/disable early stopping
- **residual** (*bool*) – Add residuals.
- **loss** (*Callable*) – Optional Custom loss function. If None, the loss is CrossEntropy for classification tasks, or RMSE for regression.

**T\_destination**

alias of TypeVar('T\_destination', bound=Dict[str, Any])

**add\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

**Parameters**

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

**apply**(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** *fn* (*Module* → None) – function to be applied to each submodule

**Returns** self

**Return type** Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
```

(continues on next page)

(continued from previous page)

```

 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)

```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

**Note:** This method modifies the module in-place.**Returns** self**Return type** Module**buffers**(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.**Yields** *torch.Tensor* – module buffer

Example:

```

>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module**cpu()** → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

**Note:** This method modifies the module in-place.**Returns** self**Return type** Module**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

**Note:** This method modifies the module in-place.

**Parameters** `device` (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**double()** → `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `double` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**dump\_patches:** `bool = False`

**eval()** → `torch.nn.modules.module.T`

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** `self`

**Return type** `Module`

**extra\_repr()** → `str`

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**fit**(*X: numpy.ndarray, y: numpy.ndarray*) → *synthcity.plugins.core.models.mlp.MLP*

**float()** → `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `float` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**forward**(*X: torch.Tensor*) → `torch.Tensor`

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**get\_buffer**(*target: str*) → torch.Tensor

Returns the buffer given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by *target*

**Return type** torch.Tensor

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by *target*

**Return type** torch.nn.Parameter

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by *target* if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
```

(continues on next page)



(continued from previous page)

```
)
)
```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by `target`

**Return type** `torch.nn.Module`

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**half()** → `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**ipu(device: Optional[Union[int, torch.device]] = None)** → `torch.nn.modules.module.T`

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device (int, optional)` – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**load\_state\_dict(state\_dict: Mapping[str, Any], strict: bool = True)**

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

**Parameters**

- **state\_dict (dict)** – a dict containing parameters and persistent buffers.

- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `True`

**Returns**

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** `NamedTuple` with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as `None` and its corresponding key exists in `state_dict`, `load_state_dict()` will raise a `RuntimeError`.

---

**modules()** → `Iterator[torch.nn.modules.module.Module]`

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, `l` will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix: str = "", recurse: bool = True*) → `Iterator[Tuple[str, torch.Tensor]]`

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

**Parameters**

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if `True`, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str*, *torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children()** → `Iterator[Tuple[str, torch.nn.modules.module.Module]]`

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str*, *Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(memo: Optional[Set[torch.nn.modules.module.Module]] = None, prefix: str = "", remove\_duplicate: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (str, Module) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(prefix: str = "", recurse: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (str) – prefix to prepend to all parameter names.
- **recurse** (bool) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (str, Parameter) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters** **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**predict**(*X: numpy.ndarray*) → numpy.ndarray

**predict\_proba**(*X: numpy.ndarray*) → numpy.ndarray

**register\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of **register\_full\_backward\_hook**() and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_buffer**(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to False. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

**Parameters**

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If None, then operations that run on buffers, such as `cuda`, are ignored. If None, the buffer is **not** included in the module's `state_dict`.
- **persistent** (*bool*) – whether the buffer is part of this module's `state_dict`.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → `torch.utils.hooks.RemovableHandle`

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook(hook)**

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module(name: str, module: Optional[torch.nn.modules.module.Module]) → None**

Alias for `add_module()`.

**register\_parameter(name: str, param: Optional[torch.nn.parameter.Parameter]) → None**

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

**Parameters**

- **name** (`str`) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (`Parameter` or `None`) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad\_(requires\_grad: bool = True) → torch.nn.modules.module.T**

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** **requires\_grad** (`bool`) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** `self`

**Return type** `Module`

**score(X: numpy.ndarray, y: numpy.ndarray) → float****set\_extra\_state(state: Any)**

This function is called from `load_state_dict()` to handle any extra state found within the `state_dict`. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its `state_dict`.

**Parameters** **state** (`dict`) – Extra state from the `state_dict`

**share\_memory()** → torch.nn.modules.module.T

See torch.Tensor.share\_memory\_()

**state\_dict**(\*args, destination=None, prefix="", keep\_vars=False)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to None are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently state\_dict() also accepts positional arguments for destination, prefix and keep\_vars in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument destination as it is not designed for end-users.

### Parameters

- **destination** (*dict*, *optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an OrderedDict will be created and returned. Default: None.
- **prefix** (*str*, *optional*) – a prefix added to parameter and buffer names to compose the keys in state\_dict. Default: ''.
- **keep\_vars** (*bool*, *optional*) – by default the Tensor's returned in the state dict are detached from autograd. If it's set to True, detaching will not be performed. Default: False.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(\*args, \*\*kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(device=None, dtype=None, non\_blocking=False)

**to**(dtype, non\_blocking=False)

**to**(tensor, non\_blocking=False)

**to**(memory\_format=torch.channels\_last)

Its signature is similar to torch.Tensor.to(), but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to dtype (if given). The integral parameters and buffers will be moved device, if that is given, but with dtypes unchanged.

When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** `self`

**Return type** `Module`

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)
```

(continues on next page)



(continued from previous page)

```

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device*: *Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (*torch.device*) – The desired device of the parameters and buffers in this module.

**Returns** *self*

**Return type** Module

**train**(*mode*: *bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (*bool*) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** *self*

**Return type** Module

**training**: *bool*

**type**(*dst\_type*: *Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (*type or string*) – the desired type

**Returns** *self*

**Return type** Module

**xpu**(*device*: *Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *device* (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**zero\_grad**(*set\_to\_none: bool = False*) → None

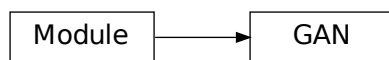
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** **set\_to\_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

## synthcity.plugins.core.models.gan module

```
class GAN(n_features: int, n_units_latent: int, n_units_conditional: int = 0, generator_n_layers_hidden: int = 2,
 generator_n_units_hidden: int = 250, generator_nonlin: str = 'leaky_relu', generator_nonlin_out:
 Optional[List[Tuple[str, int]]] = None, generator_n_iter: int = 500, generator_batch_norm: bool =
 False, generator_dropout: float = 0, generator_lr: float = 0.0002, generator_weight_decay: float =
 0.001, generator_residual: bool = True, generator_opt_betas: tuple = (0.9, 0.999),
 generator_extra_penalties: list = [], generator_extra_penalty_cbks: List[Callable] = [],
 discriminator_n_layers_hidden: int = 3, discriminator_n_units_hidden: int = 300,
 discriminator_nonlin: str = 'leaky_relu', discriminator_n_iter: int = 1, discriminator_batch_norm:
 bool = False, discriminator_dropout: float = 0.1, discriminator_lr: float = 0.0002,
 discriminator_weight_decay: float = 0.001, discriminator_opt_betas: tuple = (0.9, 0.999), batch_size:
 int = 64, random_state: int = 0, clipping_value: int = 0, lambda_gradient_penalty: float = 10,
 lambda_identifiability_penalty: float = 0.1, dataloader_sampler:
 Optional[torch.utils.data.sampler.Sampler] = None, device: Any = device(type='cpu'), n_iter_min: int
 = 100, n_iter_print: int = 10, patience: int = 20, patience_metric:
 Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None, dp_enabled: bool = False,
 dp_delta: Optional[float] = None, dp_epsilon: float = 3, dp_max_grad_norm: float = 2,
 dp_secure_mode: bool = False)
```

Bases: `torch.nn.modules.module.Module`



Basic GAN implementation.

### Parameters

- **n\_features** – int Number of features
- **n\_units\_latent** – int Number of latent units
- **n\_units\_conditional** – int Number of conditional units
- **generator\_n\_layers\_hidden** – int Number of hidden layers in the generator
- **generator\_n\_units\_hidden** – int Number of hidden units in each layer of the Generator
- **generator\_nonlin** – string, default 'elu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky\_relu'.

- **generator\_n\_iter** – int Maximum number of iterations in the Generator.
- **generator\_batch\_norm** – bool Enable/disable batch norm for the generator
- **generator\_dropout** – float Dropout value. If 0, the dropout is not used.
- **generator\_residual** – bool Use residuals for the generator
- **generator\_nonlin\_out** – Optional[List[Tuple[str, int]]] List of activations. Useful with the TabularEncoder
- **generator\_lr** – float =  $2e-4$  Generator learning rate, used by the Adam optimizer
- **generator\_weight\_decay** – float =  $1e-3$  Generator weight decay, used by the Adam optimizer
- **generator\_opt\_betas** – tuple = (0.9, 0.999) Generator initial decay rates, used by the Adam Optimizer
- **generator\_extra\_penalties** – list Additional penalties for the generator. Values: “`identifiability_penalty`”
- **generator\_extra\_penalty\_cbks** – List[Callable] Additional loss callabacks for the generator. Used by the TabularGAN for the conditional loss
- **discriminator\_n\_layers\_hidden** – int Number of hidden layers in the discriminator
- **discriminator\_n\_units\_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator\_nonlin** – string, default ‘relu’ Nonlinearity to use in the discriminator. Can be ‘elu’, ‘relu’, ‘selu’ or ‘leaky\_relu’.
- **discriminator\_n\_iter** – int Maximum number of iterations in the discriminator.
- **discriminator\_batch\_norm** – bool Enable/disable batch norm for the discriminator
- **discriminator\_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **discriminator\_lr** – float Discriminator learning rate, used by the Adam optimizer
- **discriminator\_weight\_decay** – float Discriminator weight decay, used by the Adam optimizer
- **discriminator\_opt\_betas** – tuple Initial weight decays for the Adam optimizer
- **batch\_size** – int Batch size
- **random\_state** – int random\_state used
- **clipping\_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **lambda\_gradient\_penalty** – float = 10 Weight for the gradient penalty
- **lambda\_identifiability\_penalty** – float = 0.1 Weight for the identifiability penalty, if enabled
- **dataloader\_sampler** – Optional[sampler.Sampler] Optional sampler for the dataloader, useful for conditional sampling
- **device** – Any = DEVICE CUDA/CPU
- **stopping** (# *early*) –
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.

- **n\_iter\_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before early stopping is triggered.
- **patience\_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for early stopping.
- **settings** (# *privacy*) –
- **dp\_enabled** – bool Train the discriminator with Differential Privacy guarantees
- **dp\_delta** – Optional[float] Optional DP delta: the probability of information accidentally being leaked. Usually  $1 / \text{len}(\text{dataset})$
- **dp\_epsilon** – float = 3 DP epsilon: privacy budget, which is a measure of the amount of privacy that is preserved by a given algorithm. Epsilon is a number that represents the maximum amount of information that an adversary can learn about an individual from the output of a differentially private algorithm. The smaller the value of epsilon, the more private the algorithm is. For example, an algorithm with an epsilon of 0.1 preserves more privacy than an algorithm with an epsilon of 1.0.
- **dp\_max\_grad\_norm** – float max grad norm used for gradient clipping
- **dp\_secure\_mode** – bool = False, if True uses noise generation approach robust to floating point arithmetic attacks.

**T\_destination**

alias of TypeVar('T\_destination', bound=Dict[str, Any])

**add\_module**(name: str, module: Optional[torch.nn.modules.module.Module]) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

**Parameters**

- **name** (str) – name of the child module. The child module can be accessed from this module using the given name
- **module** (Module) – child module to be added to the module.

**apply**(fn: Callable[[torch.nn.modules.module.Module], None]) → torch.nn.modules.module.T

Applies **fn** recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** **fn** (Module → None) – function to be applied to each submodule

**Returns** self

**Return type** Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
```

(continues on next page)

(continued from previous page)

```

Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)

```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**buffers**(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** *torch.Tensor* – module buffer

Example:

```

>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module

**cpu()** → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**dataloader**(*X: torch.Tensor, cond: Optional[torch.Tensor] = None*) → torch.utils.data.dataloader.DataLoader

**double**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to double datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**dump\_patches: bool = False**

**eval**() → torch.nn.modules.module.T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** self

**Return type** Module

**extra\_repr**() → str

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**fit**(*X: numpy.ndarray, cond: Optional[numpy.ndarray] = None, fake\_labels\_generator: Optional[Callable] = None, true\_labels\_generator: Optional[Callable] = None*) → [synthcity.plugins.core.models.gan.GAN](#)

**float**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to float datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**forward**(*count: int, cond: Optional[torch.Tensor] = None*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**generate**(*count: int, cond: Optional[numpy.ndarray] = None*) → numpy.ndarray

**get\_buffer**(*target: str*) → torch.Tensor

Returns the buffer given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by *target*

**Return type** torch.Tensor

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by *target*

**Return type** torch.nn.Parameter

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by *target* if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)
```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by `target`

**Return type** `torch.nn.Module`

**Raises** **`AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**`half()`** → `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**`ipu(device: Optional[Union[int, torch.device]] = None)`** → `torch.nn.modules.module.T`

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (`int`, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`



**load\_state\_dict**(*state\_dict*: Mapping[str, Any], *strict*: bool = True)

Copies parameters and buffers from *state\_dict* into this module and its descendants. If *strict* is True, then the keys of *state\_dict* must exactly match the keys returned by this module's `state_dict()` function.

#### Parameters

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in *state\_dict* match the keys returned by this module's `state_dict()` function. Default: True

#### Returns

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** NamedTuple with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as None and its corresponding key exists in *state\_dict*, `load_state_dict()` will raise a `RuntimeError`.

---

**modules**() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str*, *torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children()** → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str*, *Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo*: Optional[Set[torch.nn.modules.module.Module]] = None, *prefix*: str = "", *remove\_duplicate*: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str*, *Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str, Parameter*) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → `Iterator[torch.nn.parameter.Parameter]`

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → `torch.utils.hooks.RemovableHandle`

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_buffer**(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → `None`

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm’s `running_mean` is not a parameter, but is part of the module’s state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module’s `state_dict`.

Buffers can be accessed as attributes using given names.

**Parameters**

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as `cuda`, are ignored. If `None`, the buffer is **not** included in the module’s `state_dict`.

- **persistent** (*bool*) – whether the buffer is part of this module’s *state\_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → *torch.utils.hooks.RemovableHandle*

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The *grad\_input* and *grad\_output* are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of *grad\_input* in subsequent computations. *grad\_input* will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in *grad\_input* and *grad\_output* will be *None* for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module’s forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Alias for `add_module()`.

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad**(*requires\_grad: bool = True*) → `torch.nn.modules.module.T`

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** `requires_grad` (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** `self`

**Return type** `Module`

**set\_extra\_state**(*state: Any*)

This function is called from `load_state_dict()` to handle any extra state found within the *state\_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state\_dict*.

**Parameters** *state (dict)* – Extra state from the *state\_dict*

**share\_memory**() → torch.nn.modules.module.T

See torch.Tensor.share\_memory\_()

**state\_dict**(\*args, destination=None, prefix="", keep\_vars=False)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to None are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently `state_dict()` also accepts positional arguments for *destination*, *prefix* and *keep\_vars* in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument *destination* as it is not designed for end-users.

### Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in *state\_dict*. Default: `''`.
- **keep\_vars** (*bool, optional*) – by default the `Tensor`s returned in the *state dict* are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(\*args, \*\*kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(*device=None, dtype=None, non\_blocking=False*)

**to**(*dtype, non\_blocking=False*)

`to(tensor, non_blocking=False)`

`to(memory_format=torch.channels_last)`

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** `self`

**Return type** `Module`

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```

Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (type or string) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.



---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**zero\_grad**(*set\_to\_none: bool = False*) → `None`

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** `set_to_none` (*bool*) – instead of setting to zero, set the grads to `None`. See `torch.optim.Optimizer.zero_grad()` for details.

### **synthcity.plugins.core.models.flows module**

```
class NormalizingFlows(n_iter: int = 1000, n_layers_hidden: int = 5, n_units_hidden: int = 10, batch_size: int = 100, num_transform_blocks: int = 2, dropout: float = 0.25, batch_norm: bool = False, num_bins: int = 8, tail_bound: float = 3, lr: float = 0.001, apply_unconditional_transform: bool = True, base_distribution: str = 'standard_normal', linear_transform_type: str = 'permutation', base_transform_type: str = 'rq-autoregressive', device: Any = device(type='cpu'), n_iter_min: int = 100, n_iter_print: int = 10, patience: int = 20, patience_metric: Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None)
```

Bases: `torch.nn.modules.module.Module`

Normalizing Flows are generative models which produce tractable distributions where both sampling and density evaluation can be efficient and exact.

#### **Parameters**

- **n\_iter** – int Number of flow steps
- **n\_layers\_hidden** – int Number of transformation layers
- **n\_units\_hidden** – int Number of hidden units for each layer
- **batch\_size** – int Size of batch used for training
- **num\_transform\_blocks** – int Number of blocks to use in coupling/autoregressive nets.
- **dropout** – float Dropout probability for coupling/autoregressive nets.
- **batch\_norm** – bool Whether to use batch norm in coupling/autoregressive nets.
- **num\_bins** – int Number of bins to use for piecewise transforms.
- **tail\_bound** – float Box is on  $[-bound, bound]^2$
- **lr** – float Learning rate for optimizer.
- **apply\_unconditional\_transform** – bool Whether to unconditionally transform ‘identity’ features in the coupling layer.
- **base\_distribution** – str Possible values: “standard\_normal”
- **linear\_transform\_type** – str Type of linear transform to use. Possible values:
  - `lu` : A linear transform where we parameterize the LU decomposition of the weights.

- permutation: Permutes using a random, but fixed, permutation.
- svd: A linear module using the SVD decomposition for the weight matrix.
- **base\_transform\_type** – str Type of transform to use between linear layers. Possible values:
  - **affine-coupling** [An affine coupling layer that scales and shifts part of the variables.] Ref: L. Dinh et al., “Density estimation using Real NVP”.
  - **quadratic-coupling** : Ref: Müller et al., “Neural Importance Sampling”.
  - **rq-coupling** [Rational Quadratic Coupling] Ref: Durkan et al, “Neural Spline Flows”.
  - **affine-autoregressive :Affine Autoregressive Transform** Ref: Durkan et al, “Neural Spline Flows”.
  - **quadratic-autoregressive** [Quadratic Autoregressive Transform] Ref: Durkan et al, “Neural Spline Flows”.
  - **rq-autoregressive** [Rational Quadratic Autoregressive Transform] Ref: Durkan et al, “Neural Spline Flows”.
- **stopping (# early)** –
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.
- **n\_iter\_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before early stopping is triggered.
- **patience\_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for early stopping.

**T\_destination**

alias of TypeVar(‘T\_destination’, bound=Dict[str, Any])

**add\_module**(name: str, module: Optional[torch.nn.modules.module.Module]) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

**Parameters**

- **name** (str) – name of the child module. The child module can be accessed from this module using the given name
- **module** (Module) – child module to be added to the module.

**apply**(fn: Callable[[torch.nn.modules.module.Module], None]) → torch.nn.modules.module.T

Applies **fn** recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** **fn** (Module -> None) – function to be applied to each submodule

**Returns** self

**Return type** Module

Example:

```

>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)

```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**buffers**(recurse: bool = True) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (bool) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** torch.Tensor – module buffer

Example:

```

>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** Module – a child module

**cpu()** → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**dataloader**(*X: torch.Tensor*) → torch.utils.data.dataloader.DataLoader

**double**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to double datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**dump\_patches: bool = False**

**eval**() → torch.nn.modules.module.T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** self

**Return type** Module

**extra\_repr**() → str

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**fit**(*X: pandas.core.frame.DataFrame*) → Any

**float()** → torch.nn.modules.module.T  
 Casts all floating point parameters and buffers to float datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**forward**(*count: int*) → torch.Tensor  
 Defines the computation performed at every call.  
 Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**generate**(*count: int*) → numpy.ndarray

**get\_buffer**(*target: str*) → torch.Tensor  
 Returns the buffer given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by *target*

**Return type** torch.Tensor

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → Any  
 Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → torch.nn.parameter.Parameter  
 Returns the parameter given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by *target*

**Return type** torch.nn.Parameter

**Raises `AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**`get_submodule(target: str)`** → torch.nn.modules.module.Module

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)
```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters `target`** – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by `target`

**Return type** torch.nn.Module

**Raises `AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**`half()`** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** Module

**`ipu(device: Optional[Union[int, torch.device]] = None)`** → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**load\_state\_dict**(*state\_dict: Mapping[str, Any]*, *strict: bool = True*)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

**Parameters**

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `True`

**Returns**

- **missing\_keys** is a list of `str` containing the missing keys
- **unexpected\_keys** is a list of `str` containing the unexpected keys

**Return type** `NamedTuple` with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as `None` and its corresponding key exists in `state_dict`, `load_state_dict()` will raise a `RuntimeError`.

---

**modules**() → `Iterator[torch.nn.modules.module.Module]`

Returns an iterator over all modules in the network.

**Yields** `Module` – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, `l` will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix: str = ""*, *recurse: bool = True*) → `Iterator[Tuple[str, torch.Tensor]]`

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

**Parameters**

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if `True`, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str*, *torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children()** → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str*, *Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo*: Optional[Set[torch.nn.modules.module.Module]] = None, *prefix*: str = "", *remove\_duplicate*: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str*, *Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.



### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str*, *Parameter*) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_buffer**(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm’s `running_mean` is not a parameter, but is part of the module’s state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to False. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module’s `state_dict`.

Buffers can be accessed as attributes using given names.

### Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name

- **tensor** (*Tensor or None*) – buffer to be registered. If *None*, then operations that run on buffers, such as *cuda*, are ignored. If *None*, the buffer is **not** included in the module's *state\_dict*.
- **persistent** (*bool*) – whether the buffer is part of this module's *state\_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the *forward*. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the *forward*. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → *torch.utils.hooks.RemovableHandle*

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The *grad\_input* and *grad\_output* are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of *grad\_input* in subsequent computations. *grad\_input* will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in *grad\_input* and *grad\_output* will be *None* for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The module argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Alias for `add_module()`.

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad\_**(*requires\_grad: bool = True*) → `torch.nn.modules.module.T`

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See `locally-disable-grad-doc` for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** `requires_grad` (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** self

**Return type** Module

**set\_extra\_state**(*state: Any*)

This function is called from `load_state_dict()` to handle any extra state found within the *state\_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state\_dict*.

**Parameters** **state** (*dict*) – Extra state from the *state\_dict*

**share\_memory**() → torch.nn.modules.module.T

See `torch.Tensor.share_memory_()`

**state\_dict**(\*args, *destination=None, prefix="", keep\_vars=False*)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently `state_dict()` also accepts positional arguments for *destination*, *prefix* and *keep\_vars* in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument *destination* as it is not designed for end-users.

### Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in *state\_dict*. Default: `''`.
- **keep\_vars** (*bool, optional*) – by default the `Tensor`s returned in the *state dict* are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(\*args, \*\*kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

```
to(device=None, dtype=None, non_blocking=False)
```

```
to(dtype, non_blocking=False)
```

```
to(tensor, non_blocking=False)
```

```
to(memory_format=torch.channels_last)
```

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** `self`

**Return type** `Module`

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
```

(continues on next page)

(continued from previous page)

```

 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (type or string) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device (int, optional)` – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**zero\_grad**(*set\_to\_none: bool = False*) → `None`

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** `set_to_none (bool)` – instead of setting to zero, set the grads to `None`. See `torch.optim.Optimizer.zero_grad()` for details.

**create\_alternating\_binary\_mask**(*features: int, even: bool = True*) → `torch.Tensor`

Creates a binary mask of a given dimension which alternates its masking. :param features: Dimension of mask. :param even: If True, even values are assigned 1s, odd 0s. If False, vice versa. :return: Alternating binary mask of type `torch.Tensor`.

## synthcity.plugins.core.models.vae module

**class Decoder**(*n\_units\_embedding: int, n\_units\_out: int, n\_layers\_hidden: int = 1, n\_units\_hidden: int = 100, nonlin: str = 'relu', nonlin\_out: Optional[List[Tuple[str, int]]] = None, random\_state: int = 0, dropout: float = 0.1, batch\_norm: bool = True, residual: bool = False, device: Any = device(type='cpu')*)

Bases: `torch.nn.modules.module.Module`

**T\_destination**

alias of `TypeVar('T_destination', bound=Dict[str, Any])`

**add\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → `None`

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

**Parameters**

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

**apply**(*fn: Callable[[torch.nn.modules.module.Module], None]*) → `torch.nn.modules.module.T`

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as `self`. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** `fn (Module -> None)` – function to be applied to each submodule

**Returns** `self`

**Return type** `Module`

Example:

```

>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)

```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**buffers**(recurse: bool = True) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (bool) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** torch.Tensor – module buffer

Example:

```

>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** Module – a child module

**cpu()** → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.



---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**double**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to double datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**dump\_patches: bool = False**

**eval**() → torch.nn.modules.module.T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** self

**Return type** Module

**extra\_repr**() → str

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**float**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to float datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**forward**(*X: torch.Tensor, cond: Optional[torch.Tensor] = None*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**get\_buffer**(*target: str*) → torch.Tensor

Returns the buffer given by **target** if it exists, otherwise throws an error.

See the docstring for **get\_submodule** for a more detailed explanation of this method's functionality as well as how to correctly specify **target**.

**Parameters** **target** – The fully-qualified string name of the buffer to look for. (See **get\_submodule** for how to specify a fully-qualified string.)

**Returns** The buffer referenced by **target**

**Return type** torch.Tensor

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by **target** if it exists, otherwise throws an error.

See the docstring for **get\_submodule** for a more detailed explanation of this method's functionality as well as how to correctly specify **target**.

**Parameters** **target** – The fully-qualified string name of the Parameter to look for. (See **get\_submodule** for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by **target**

**Return type** torch.nn.Parameter

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by **target** if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)
```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in **target**. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters target** – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by **target**

**Return type** torch.nn.Module

**Raises AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**half**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**ipu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**load\_state\_dict**(*state\_dict*: Mapping[str, Any], *strict*: bool = True)

Copies parameters and buffers from *state\_dict* into this module and its descendants. If *strict* is True, then the keys of *state\_dict* must exactly match the keys returned by this module's `state_dict()` function.

**Parameters**

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in *state\_dict* match the keys returned by this module's `state_dict()` function. Default: True

**Returns**

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** NamedTuple with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as `None` and its corresponding key exists in *state\_dict*, `load_state_dict()` will raise a `RuntimeError`.

---

**modules**() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

**Parameters**

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str*, *torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children()** → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str*, *Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo*: Optional[Set[torch.nn.modules.module.Module]] = None, *prefix*: str = "", *remove\_duplicate*: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str*, *Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str*, *Parameter*) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → `Iterator[torch.nn.parameter.Parameter]`

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → `torch.utils.hooks.RemovableHandle`

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_buffer**(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → `None`

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm’s `running_mean` is not a parameter, but is part of the module’s state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module’s `state_dict`.

Buffers can be accessed as attributes using given names.

**Parameters**

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as `cuda`, are ignored. If `None`, the buffer is **not** included in the module’s `state_dict`.

- **persistent** (*bool*) – whether the buffer is part of this module’s *state\_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → *torch.utils.hooks.RemovableHandle*

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The *grad\_input* and *grad\_output* are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of *grad\_input* in subsequent computations. *grad\_input* will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in *grad\_input* and *grad\_output* will be *None* for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module’s forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Alias for `add_module()`.

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad\_**(*requires\_grad: bool = True*) → `torch.nn.modules.module.T`

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** `requires_grad` (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** `self`

**Return type** `Module`



**set\_extra\_state**(*state: Any*)

This function is called from `load_state_dict()` to handle any extra state found within the *state\_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state\_dict*.

**Parameters** *state (dict)* – Extra state from the *state\_dict*

**share\_memory**() → torch.nn.modules.module.T

See `torch.Tensor.share_memory_()`

**state\_dict**(\*args, destination=None, prefix="", keep\_vars=False)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently `state_dict()` also accepts positional arguments for *destination*, *prefix* and *keep\_vars* in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument *destination* as it is not designed for end-users.

### Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in *state\_dict*. Default: `''`.
- **keep\_vars** (*bool, optional*) – by default the `Tensor`s returned in the *state dict* are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(\*args, \*\*kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(*device=None, dtype=None, non\_blocking=False*)

**to**(*dtype, non\_blocking=False*)

`to(tensor, non_blocking=False)`

`to(memory_format=torch.channels_last)`

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** `self`

**Return type** `Module`

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```

Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (type or string) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**zero\_grad**(*set\_to\_none: bool = False*) → `None`

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** `set_to_none` (*bool*) – instead of setting to zero, set the grads to `None`. See `torch.optim.Optimizer.zero_grad()` for details.

**class Encoder**(*n\_units\_in: int, n\_units\_embedding: int, n\_layers\_hidden: int = 1, n\_units\_hidden: int = 100, nonlin: str = 'relu', random\_state: int = 0, dropout: float = 0.1, batch\_norm: bool = True, residual: bool = False, device: Any = device(type='cpu')*)

Bases: `torch.nn.modules.module.Module`

**T\_destination**

alias of `TypeVar('T_destination', bound=Dict[str, Any])`

**add\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → `None`

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

**Parameters**

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

**apply**(*fn: Callable[[torch.nn.modules.module.Module], None]*) → `torch.nn.modules.module.T`

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as `self`. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** `fn` (*Module -> None*) – function to be applied to each submodule

**Returns** `self`

**Return type** `Module`

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.]
```

(continues on next page)

(continued from previous page)

```

 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**buffers**(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** *torch.Tensor* – module buffer

Example:

```

>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module

**cpu()** → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** *self*

**Return type** *Module*

**double()** → *torch.nn.modules.module.T*

Casts all floating point parameters and buffers to `double` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** *Module*

**dump\_patches:** `bool = False`

**eval()** → *torch.nn.modules.module.T*

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** *self*

**Return type** *Module*

**extra\_repr()** → *str*

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**float()** → *torch.nn.modules.module.T*

Casts all floating point parameters and buffers to `float` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** *Module*

**forward**(*X: torch.Tensor*, *cond: Optional[torch.Tensor] = None*) → *Tuple[torch.Tensor, torch.Tensor]*

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**get\_buffer**(*target: str*) → torch.Tensor

Returns the buffer given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by *target*

**Return type** torch.Tensor

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by *target*

**Return type** torch.nn.Parameter

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by *target* if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
```

(continues on next page)

(continued from previous page)

```
)
)
```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by `target`

**Return type** `torch.nn.Module`

**Raises** **`AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**`half()`**  $\rightarrow$  `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**`ipu(device: Optional[Union[int, torch.device]] = None)`**  $\rightarrow$  `torch.nn.modules.module.T`

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device (int, optional)` – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**`load_state_dict(state_dict: Mapping[str, Any], strict: bool = True)`**

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

**Parameters**

- **`state_dict (dict)`** – a dict containing parameters and persistent buffers.



- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `True`

#### Returns

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** `NamedTuple` with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as `None` and its corresponding key exists in `state_dict`, `load_state_dict()` will raise a `RuntimeError`.

---

**modules()** → `Iterator[torch.nn.modules.module.Module]`

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, `l` will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix: str = "", recurse: bool = True*) → `Iterator[Tuple[str, torch.Tensor]]`

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if `True`, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str*, *torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children()** → `Iterator[Tuple[str, torch.nn.modules.module.Module]]`

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str*, *Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(memo: Optional[Set[torch.nn.modules.module.Module]] = None, prefix: str = "", remove\_duplicate: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (str, Module) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(prefix: str = "", recurse: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (str) – prefix to prepend to all parameter names.
- **recurse** (bool) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (str, Parameter) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters** **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_buffer**(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to False. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

**Parameters**

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If None, then operations that run on buffers, such as `cuda`, are ignored. If None, the buffer is **not** included in the module's `state_dict`.
- **persistent** (*bool*) – whether the buffer is part of this module's `state_dict`.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → `torch.utils.hooks.RemovableHandle`

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Alias for `add_module()`.

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad**(*requires\_grad: bool = True*) → `torch.nn.modules.module.T`

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See `locally-disable-grad-doc` for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** **requires\_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** `self`

**Return type** `Module`

**set\_extra\_state**(*state: Any*)

This function is called from `load_state_dict()` to handle any extra state found within the `state_dict`. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its `state_dict`.

**Parameters** **state** (*dict*) – Extra state from the `state_dict`

**share\_memory**() → `torch.nn.modules.module.T`

See `torch.Tensor.share_memory_()`

**state\_dict**(\*args, destination=None, prefix="", keep\_vars=False)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to None are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument `destination` as it is not designed for end-users.

### Parameters

- **destination** (*dict*, *optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str*, *optional*) – a prefix added to parameter and buffer names to compose the keys in `state_dict`. Default: `''`.
- **keep\_vars** (*bool*, *optional*) – by default the `Tensor`s returned in the state dict are detached from `autograd`. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(\*args, \*\*kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(device=None, dtype=None, non\_blocking=False)

**to**(dtype, non\_blocking=False)

**to**(tensor, non\_blocking=False)

**to**(memory\_format=torch.channels\_last)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** `self`

**Return type** `Module`

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
```

(continues on next page)

(continued from previous page)

```

tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device*: Union[str, torch.device]) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode*: bool = True) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training**: bool

**type**(*dst\_type*: Union[torch.dtype, str]) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (type or string) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device*: Optional[Union[int, torch.device]] = None) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *device* (int, optional) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module



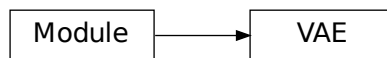
**zero\_grad**(*set\_to\_none: bool = False*) → None

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** `set_to_none` (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

```
class VAE(n_features: int, n_units_embedding: int, n_units_conditional: int = 0, batch_size: int = 100, n_iter: int = 500, random_state: int = 0, lr: float = 0.0002, weight_decay: float = 0.001, decoder_n_layers_hidden: int = 2, decoder_n_units_hidden: int = 250, decoder_nonlin: str = 'leaky_relu', decoder_nonlin_out: Optional[List[Tuple[str, int]]] = None, decoder_batch_norm: bool = False, decoder_dropout: float = 0, decoder_residual: bool = True, encoder_n_layers_hidden: int = 3, encoder_n_units_hidden: int = 300, encoder_nonlin: str = 'leaky_relu', encoder_batch_norm: bool = False, encoder_dropout: float = 0.1, loss_strategy: str = 'standard', loss_factor: int = 2, robust_divergence_beta: int = 2, dataloader_sampler: Optional[torch.utils.data.sampler.Sampler] = None, device: Any = device(type='cpu'), extra_loss_cbks: List[Callable] = [], clipping_value: int = 1, n_iter_min: int = 100, n_iter_print: int = 10, patience: int = 20)
```

Bases: `torch.nn.modules.module.Module`



Basic VAE implementation.

#### Parameters

- **n\_features** – int Number of features in the dataset
- **n\_units\_embedding** – int Number of units in the latent space
- **batch\_size** – int Training batch size
- **n\_iter** – int Number of training iterations
- **random\_state** – int Random random\_state
- **lr** – float Learning rate
- **weight\_decay** – float: Optimizer weight decay
- **decoder\_n\_layers\_hidden** – int Number of hidden layers in the decoder
- **decoder\_n\_units\_hidden** – int Number of units in the hidden layer in the decoder
- **decoder\_nonlin\_out** – List List of activations layout, as generated by the tabular encoder
- **decoder\_batch\_norm** – bool Use batchnorm in the decoder
- **decoder\_dropout** – float Use dropout in the decoder
- **decoder\_residual** – bool Use residual connections in the decoder
- **encoder\_n\_layers\_hidden** – int Number of hidden layers in the encoder
- **encoder\_n\_units\_hidden** – int Number of units in the hidden layer in the encoder
- **encoder\_batch\_norm** – bool Use batchnorm in the encoder

- **encoder\_dropout** – float Use dropout in the encoder
- **encoder\_residual** – bool Use residual connections in the encoder
- **loss\_strategy** – str - standard: classic VAE loss - robust\_divergence: Algorithm 1 in “Robust Variational Autoencoder for Tabular Data with Divergence”
- **loss\_factor** – int Parameter for the standard loss
- **robust\_divergence\_beta** – int Parameter for the robust\_divergence loss
- **dataloader\_sampler** – Custom sampler used by the dataloader, useful for conditional sampling.
- **device** – CPU/CUDA
- **extra\_loss\_cbks** – Custom loss callbacks. For example, for conditional loss.
- **clipping\_value** – Gradients clipping value. Zero disables the feature
- **stopping** (# early) –
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.
- **n\_iter\_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before early stopping is triggered.

**T\_destination**

alias of `TypeVar('T_destination', bound=Dict[str, Any])`

**add\_module**(name: str, module: Optional[torch.nn.modules.module.Module]) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

**Parameters**

- **name** (str) – name of the child module. The child module can be accessed from this module using the given name
- **module** (Module) – child module to be added to the module.

**apply**(fn: Callable[[torch.nn.modules.module.Module], None]) → torch.nn.modules.module.T

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** `fn` (Module → None) – function to be applied to each submodule

**Returns** self

**Return type** Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
```

(continues on next page)

(continued from previous page)

```
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**buffers**(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module

**cpu()** → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**double**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to double datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**dump\_patches: bool = False**

**eval**() → torch.nn.modules.module.T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** self

**Return type** Module

**extra\_repr**() → str

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**fit**(*X: numpy.ndarray, cond: Optional[numpy.ndarray] = None*) → Any

**float**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to float datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**forward**(\*input: Any) → None

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**generate**(count: int, cond: Optional[numpy.ndarray] = None) → numpy.ndarray

**get\_buffer**(target: str) → torch.Tensor

Returns the buffer given by **target** if it exists, otherwise throws an error.

See the docstring for **get\_submodule** for a more detailed explanation of this method's functionality as well as how to correctly specify **target**.

**Parameters target** – The fully-qualified string name of the buffer to look for. (See **get\_submodule** for how to specify a fully-qualified string.)

**Returns** The buffer referenced by **target**

**Return type** torch.Tensor

**Raises AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(target: str) → torch.nn.parameter.Parameter

Returns the parameter given by **target** if it exists, otherwise throws an error.

See the docstring for **get\_submodule** for a more detailed explanation of this method's functionality as well as how to correctly specify **target**.

**Parameters target** – The fully-qualified string name of the Parameter to look for. (See **get\_submodule** for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by **target**

**Return type** torch.nn.Parameter

**Raises AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(target: str) → torch.nn.modules.module.Module

Returns the submodule given by **target** if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```

A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)

```

(The diagram shows an `nn.Module` A. A has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by `target`

**Return type** `torch.nn.Module`

**Raises** **`AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**half()** → `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**ipu(device: Optional[Union[int, torch.device]] = None)** → `torch.nn.modules.module.T`

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (`int`, optional) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**load\_state\_dict**(*state\_dict*: Mapping[str, Any], *strict*: bool = True)

Copies parameters and buffers from *state\_dict* into this module and its descendants. If *strict* is True, then the keys of *state\_dict* must exactly match the keys returned by this module's `state_dict()` function.

#### Parameters

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in *state\_dict* match the keys returned by this module's `state_dict()` function. Default: True

#### Returns

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** NamedTuple with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as None and its corresponding key exists in *state\_dict*, `load_state_dict()` will raise a `RuntimeError`.

---

**modules**() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str*, *torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children()** → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str*, *Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo*: Optional[Set[torch.nn.modules.module.Module]] = None, *prefix*: str = "", *remove\_duplicate*: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str*, *Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.



- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str, Parameter*) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → `Iterator[torch.nn.parameter.Parameter]`

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → `torch.utils.hooks.RemovableHandle`

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_buffer**(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → `None`

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm’s `running_mean` is not a parameter, but is part of the module’s state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module’s `state_dict`.

Buffers can be accessed as attributes using given names.

**Parameters**

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as `cuda`, are ignored. If `None`, the buffer is **not** included in the module’s `state_dict`.

- **persistent** (*bool*) – whether the buffer is part of this module’s *state\_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → *torch.utils.hooks.RemovableHandle*

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The *grad\_input* and *grad\_output* are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of *grad\_input* in subsequent computations. *grad\_input* will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in *grad\_input* and *grad\_output* will be *None* for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module’s forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Alias for `add_module()`.

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad**(*requires\_grad: bool = True*) → `torch.nn.modules.module.T`

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** `requires_grad` (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** `self`

**Return type** `Module`

**set\_extra\_state**(*state: Any*)

This function is called from `load_state_dict()` to handle any extra state found within the *state\_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state\_dict*.

**Parameters** *state (dict)* – Extra state from the *state\_dict*

**share\_memory**() → torch.nn.modules.module.T

See `torch.Tensor.share_memory_()`

**state\_dict**(\*args, destination=None, prefix="", keep\_vars=False)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently `state_dict()` also accepts positional arguments for *destination*, *prefix* and *keep\_vars* in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument *destination* as it is not designed for end-users.

### Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in *state\_dict*. Default: `''`.
- **keep\_vars** (*bool, optional*) – by default the `Tensor`s returned in the *state dict* are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(\*args, \*\*kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(*device=None, dtype=None, non\_blocking=False*)

**to**(*dtype, non\_blocking=False*)

`to(tensor, non_blocking=False)`

`to(memory_format=torch.channels_last)`

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** `self`

**Return type** `Module`

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```

Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (type or string) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**zero\_grad**(*set\_to\_none: bool = False*) → `None`

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** `set_to_none` (*bool*) – instead of setting to zero, set the grads to `None`. See `torch.optim.Optimizer.zero_grad()` for details.

## synthcity.plugins.core.models.transformer module

**class TransformerModel**(*n\_units\_in: int*, *n\_units\_hidden: int = 64*, *n\_head: int = 1*, *d\_ffn: int = 128*, *dropout: float = 0.1*, *activation: str = 'relu'*, *n\_layers\_hidden: int = 1*, *device: Any = device(type='cpu')*)

Bases: `torch.nn.modules.module.Module`

**T\_destination**

alias of `TypeVar('T_destination', bound=Dict[str, Any])`

**add\_module**(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → `None`

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

**Parameters**

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

**apply**(*fn: Callable[[torch.nn.modules.module.Module], None]*) → `torch.nn.modules.module.T`

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as `self`. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** `fn` (*Module -> None*) – function to be applied to each submodule

**Returns** `self`

**Return type** `Module`

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
```

(continues on next page)

(continued from previous page)

```

Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)

```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**buffers**(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** *torch.Tensor* – module buffer

Example:

```

>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module

**cpu()** → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module



**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** *self*

**Return type** Module

**double**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to double datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** Module

**dump\_patches: bool = False**

**eval**() → torch.nn.modules.module.T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** *self*

**Return type** Module

**extra\_repr**() → str

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**float**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to float datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** Module

**forward**(*x: torch.Tensor*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**get\_buffer**(*target: str*) → torch.Tensor

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

**Parameters** `target` – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by `target`

**Return type** torch.Tensor

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

**Parameters** `target` – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by `target`

**Return type** torch.nn.Parameter

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 ...
)
)
```

(continues on next page)

(continued from previous page)

```

 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)

```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by `target`

**Return type** `torch.nn.Module`

**Raises** **`AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**`half()`**  $\rightarrow$  `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**`ipu(device: Optional[Union[int, torch.device]] = None)`**  $\rightarrow$  `torch.nn.modules.module.T`

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (`int`, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**load\_state\_dict**(*state\_dict*: Mapping[str, Any], *strict*: bool = True)

Copies parameters and buffers from *state\_dict* into this module and its descendants. If *strict* is True, then the keys of *state\_dict* must exactly match the keys returned by this module's *state\_dict*() function.

#### Parameters

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in *state\_dict* match the keys returned by this module's *state\_dict*() function. Default: True

#### Returns

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** NamedTuple with *missing\_keys* and *unexpected\_keys* fields

---

**Note:** If a parameter or buffer is registered as None and its corresponding key exists in *state\_dict*, *load\_state\_dict*() will raise a *RuntimeError*.

---

**modules**() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str*, *torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children()** → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str*, *Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo*: Optional[Set[torch.nn.modules.module.Module]] = None, *prefix*: str = "", *remove\_duplicate*: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str*, *Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str*, *Parameter*) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → `Iterator[torch.nn.parameter.Parameter]`

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → `torch.utils.hooks.RemovableHandle`

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_buffer**(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → `None`

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm’s `running_mean` is not a parameter, but is part of the module’s state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to False. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module’s `state_dict`.

Buffers can be accessed as attributes using given names.

**Parameters**

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If None, then operations that run on buffers, such as `cuda`, are ignored. If None, the buffer is **not** included in the module’s `state_dict`.

- **persistent** (*bool*) – whether the buffer is part of this module’s *state\_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → *torch.utils.hooks.RemovableHandle*

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The *grad\_input* and *grad\_output* are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of *grad\_input* in subsequent computations. *grad\_input* will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in *grad\_input* and *grad\_output* will be *None* for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module’s forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Alias for `add_module()`.

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad\_**(*requires\_grad: bool = True*) → `torch.nn.modules.module.T`

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** `requires_grad` (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** `self`

**Return type** `Module`



**set\_extra\_state**(*state: Any*)

This function is called from `load_state_dict()` to handle any extra state found within the *state\_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state\_dict*.

**Parameters** *state (dict)* – Extra state from the *state\_dict*

**share\_memory**() → torch.nn.modules.module.T

See `torch.Tensor.share_memory_()`

**state\_dict**(\*args, *destination=None, prefix="", keep\_vars=False*)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently `state_dict()` also accepts positional arguments for *destination*, *prefix* and *keep\_vars* in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument *destination* as it is not designed for end-users.

### Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in *state\_dict*. Default: `''`.
- **keep\_vars** (*bool, optional*) – by default the `Tensor`s returned in the *state dict* are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(\*args, \*\*kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(*device=None, dtype=None, non\_blocking=False*)

**to**(*dtype, non\_blocking=False*)

`to(tensor, non_blocking=False)`

`to(memory_format=torch.channels_last)`

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** `self`

**Return type** `Module`

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```

Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (type or string) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**zero\_grad**(*set\_to\_none: bool = False*) → `None`

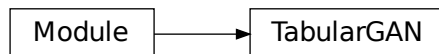
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** `set_to_none` (*bool*) – instead of setting to zero, set the grads to `None`. See `torch.optim.Optimizer.zero_grad()` for details.

### `synthcity.plugins.core.models.tabular_gan` module

**class** `TabularGAN`(*X: pandas.core.frame.DataFrame, n\_units\_latent: int, cond: Optional[Union[pandas.core.frame.DataFrame, pandas.core.series.Series, numpy.ndarray]] = None, generator\_n\_layers\_hidden: int = 2, generator\_n\_units\_hidden: int = 150, generator\_nonlin: str = 'leaky\_relu', generator\_nonlin\_out\_discrete: str = 'softmax', generator\_nonlin\_out\_continuous: str = 'none', generator\_n\_iter: int = 1000, generator\_batch\_norm: bool = False, generator\_dropout: float = 0.01, generator\_lr: float = 0.001, generator\_weight\_decay: float = 0.001, generator\_opt\_betas: tuple = (0.9, 0.999), generator\_residual: bool = True, generator\_extra\_penalties: list = [], discriminator\_n\_layers\_hidden: int = 3, discriminator\_n\_units\_hidden: int = 300, discriminator\_nonlin: str = 'leaky\_relu', discriminator\_n\_iter: int = 1, discriminator\_batch\_norm: bool = False, discriminator\_dropout: float = 0.1, discriminator\_lr: float = 0.001, discriminator\_weight\_decay: float = 0.001, discriminator\_opt\_betas: tuple = (0.9, 0.999), batch\_size: int = 64, random\_state: int = 0, clipping\_value: int = 0, lambda\_gradient\_penalty: float = 10, lambda\_identifiability\_penalty: float = 0.1, encoder\_max\_clusters: int = 20, encoder: Any = None, encoder\_whitelist: list = [], dataloader\_sampler: Optional[synthcity.utils.samplers.BaseSampler] = None, device: Any = device(type='cpu'), patience: int = 10, patience\_metric: Optional[synthcity.metrics.weighted\_metrics.WeightedMetrics] = None, n\_iter\_print: int = 50, n\_iter\_min: int = 100, adjust\_inference\_sampling: bool = False, dp\_enabled: bool = False, dp\_epsilon: float = 3, dp\_delta: Optional[float] = None, dp\_max\_grad\_norm: float = 2, dp\_secure\_mode: bool = False)*

Bases: `torch.nn.modules.module.Module`



GAN for tabular data.

This class combines GAN and tabular encoder to form a generative model for tabular data.

## Parameters

- **X** – `pd.DataFrame` Reference dataset, used for training the tabular encoder
- **n\_units\_latent** – int Number of latent units
- **cond** – Optional Optional conditional
- **generator\_n\_layers\_hidden** – int Number of hidden layers in the generator
- **generator\_n\_units\_hidden** – int Number of hidden units in each layer of the Generator
- **generator\_nonlin** – string, default ‘elu’ Nonlinearity to use in the generator. Can be ‘elu’, ‘relu’, ‘selu’ or ‘leaky\_relu’.
- **generator\_n\_iter** – int Maximum number of iterations in the Generator.
- **generator\_batch\_norm** – bool Enable/disable batch norm for the generator
- **generator\_dropout** – float Dropout value. If 0, the dropout is not used.
- **generator\_residual** – bool Use residuals for the generator
- **generator\_nonlin\_out** – Optional[List[Tuple[str, int]]] List of activations. Useful with the TabularEncoder
- **generator\_lr** – float =  $2e-4$  Generator learning rate, used by the Adam optimizer
- **generator\_weight\_decay** – float =  $1e-3$  Generator weight decay, used by the Adam optimizer
- **generator\_opt\_betas** – tuple = (0.9, 0.999) Generator initial decay rates, used by the Adam Optimizer
- **generator\_extra\_penalties** – list Additional penalties for the generator. Values: “identifiability\_penalty”
- **generator\_extra\_penalty\_cbks** – List[Callable] Additional loss callabacks for the generator. Used by the TabularGAN for the conditional loss
- **discriminator\_n\_layers\_hidden** – int Number of hidden layers in the discriminator
- **discriminator\_n\_units\_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator\_nonlin** – string, default ‘relu’ Nonlinearity to use in the discriminator. Can be ‘elu’, ‘relu’, ‘selu’ or ‘leaky\_relu’.
- **discriminator\_n\_iter** – int Maximum number of iterations in the discriminator.
- **discriminator\_batch\_norm** – bool Enable/disable batch norm for the discriminator
- **discriminator\_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **discriminator\_lr** – float Discriminator learning rate, used by the Adam optimizer
- **discriminator\_weight\_decay** – float Discriminator weight decay, used by the Adam optimizer
- **discriminator\_opt\_betas** – tuple Initial weight decays for the Adam optimizer
- **batch\_size** – int Batch size
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.

- **random\_state** – int random\_state used
- **n\_iter\_min** – int Minimum number of iterations to go through before starting early stopping
- **clipping\_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **lambda\_gradient\_penalty** – float = 10 Weight for the gradient penalty
- **lambda\_identifiability\_penalty** – float = 0.1 Weight for the identifiability penalty, if enabled
- **dataloader\_sampler** – Optional[sampler.Sampler] Optional sampler for the dataloader, useful for conditional sampling
- **device** – Any = DEVICE CUDA/CPU
- **adjust\_inference\_sampling** – bool Adjust the marginal probabilities in the synthetic data to closer match the training set. Active only with the ConditionalSampler
- **settings** (# *privacy*) –
- **dp\_enabled** – bool Train the discriminator with Differential Privacy guarantees
- **dp\_delta** – Optional[float] Optional DP delta: the probability of information accidentally being leaked. Usually 1 / len(dataset)
- **dp\_epsilon** – float = 3 DP epsilon: privacy budget, which is a measure of the amount of privacy that is preserved by a given algorithm. Epsilon is a number that represents the maximum amount of information that an adversary can learn about an individual from the output of a differentially private algorithm. The smaller the value of epsilon, the more private the algorithm is. For example, an algorithm with an epsilon of 0.1 preserves more privacy than an algorithm with an epsilon of 1.0.
- **dp\_max\_grad\_norm** – float max grad norm used for gradient clipping
- **dp\_secure\_mode** – bool = False, if True uses noise generation approach robust to floating point arithmetic attacks.
- **encoder\_max\_clusters** – int The max number of clusters to create for continuous columns when encoding
- **encoder** – Pre-trained tabular encoder. If None, a new encoder is trained.
- **encoder\_whitelist** – Ignore columns from encoding

### **T\_destination**

alias of TypeVar('T\_destination', bound=Dict[str, Any])

**add\_module**(name: str, module: Optional[torch.nn.modules.module.Module]) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

### **Parameters**

- **name** (str) – name of the child module. The child module can be accessed from this module using the given name
- **module** (Module) – child module to be added to the module.

**adjust\_inference\_sampling**(enabled: bool) → None

**apply**(fn: Callable[[torch.nn.modules.module.Module], None]) → torch.nn.modules.module.T

Applies fn recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

**Parameters** `fn` (Module -> None) – function to be applied to each submodule

**Returns** `self`

**Return type** Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** Module

**buffers** (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module

**cpu()** → torch.nn.modules.module.T  
Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** *Module*

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T  
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** *self*

**Return type** *Module*

**decode**(*X: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

**double**() → torch.nn.modules.module.T  
Casts all floating point parameters and buffers to double datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** *Module*

**dump\_patches:** **bool = False**

**encode**(*X: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

**eval**() → torch.nn.modules.module.T  
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** *self*

**Return type** *Module*



**extra\_repr()** → str

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**fit**(*X*: *pandas.core.frame.DataFrame*, *cond*: *Optional[Union[pandas.core.frame.DataFrame, pandas.core.series.Series, numpy.ndarray]] = None*, *fake\_labels\_generator*: *Optional[Callable] = None*, *true\_labels\_generator*: *Optional[Callable] = None*, *encoded*: *bool = False*) → Any

**float()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to float datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**forward**(*count*: int, *cond*: *Optional[Union[pandas.core.frame.DataFrame, numpy.ndarray]] = None*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**generate**(*count*: int, *cond*: *Optional[Union[pandas.core.frame.DataFrame, pandas.core.series.Series, numpy.ndarray]] = None*) → pandas.core.frame.DataFrame

**get\_buffer**(*target*: str) → torch.Tensor

Returns the buffer given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by *target*

**Return type** torch.Tensor

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_encoder()** → *synthcity.plugins.core.models.tabular\_encoder.TabularEncoder*

**get\_extra\_state()** → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by *target*

**Return type** torch.nn.Parameter

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by *target* if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)
```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in *target*. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** *target* – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by *target*

**Return type** torch.nn.Module

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**half**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** Module

**ipu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**load\_state\_dict**(*state\_dict: Mapping[str, Any], strict: bool = True*)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is True, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

**Parameters**

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: True

**Returns**

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** NamedTuple with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as `None` and its corresponding key exists in `state_dict`, `load_state_dict()` will raise a `RuntimeError`.

---

**modules**() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, `l` will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
```

(continues on next page)

(continued from previous page)

```
(0): Linear(in_features=2, out_features=2, bias=True)
(1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str, torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children**() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str, Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo: Optional[Set[torch.nn.modules.module.Module]] = None, prefix: str = "", remove\_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str, Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```

>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))

```

**named\_parameters**(*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str, Parameter*) – Tuple containing the name and parameter

Example:

```

>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())

```

**parameters**(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```

>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

**register\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_buffer**(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

#### Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as `cuda`, are ignored. If `None`, the buffer is **not** included in the module's `state_dict`.
- **persistent** (*bool*) – whether the buffer is part of this module's `state_dict`.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

```
register_full_backward_hook(hook: Callable[[torch.nn.modules.module.Module,
 Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor,
 ...], torch.Tensor]], Union[None, torch.Tensor]]) →
torch.utils.hooks.RemovableHandle
```

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

```
register_load_state_dict_post_hook(hook)
```

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

```
register_module(name: str, module: Optional[torch.nn.modules.module.Module]) → None
```

Alias for `add_module()`.

```
register_parameter(name: str, param: Optional[torch.nn.parameter.Parameter]) → None
```

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

**Parameters**

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If *None*, then operations that run on parameters, such as [cuda](#), are ignored. If *None*, the parameter is **not** included in the module's *state\_dict*.

**requires\_grad\_**(*requires\_grad: bool = True*) → torch.nn.modules.module.T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' *requires\_grad* attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See [locally-disable-grad-doc](#) for a comparison between *.requires\_grad\_()* and several similar mechanisms that may be confused with it.

**Parameters** *requires\_grad* (*bool*) – whether autograd should record operations on parameters in this module. Default: *True*.

**Returns** *self*

**Return type** Module

**set\_extra\_state**(*state: Any*)

This function is called from [load\\_state\\_dict\(\)](#) to handle any extra state found within the *state\_dict*. Implement this function and a corresponding [get\\_extra\\_state\(\)](#) for your module if you need to store extra state within its *state\_dict*.

**Parameters** *state* (*dict*) – Extra state from the *state\_dict*

**share\_memory**() → torch.nn.modules.module.T

See [torch.Tensor.share\\_memory\\_\(\)](#)

**state\_dict**(\**args*, *destination=None*, *prefix=""*, *keep\_vars=False*)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to *None* are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently *state\_dict()* also accepts positional arguments for *destination*, *prefix* and *keep\_vars* in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument *destination* as it is not designed for end-users.

### Parameters

- **destination** (*dict*, *optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an *OrderedDict* will be created and returned. Default: *None*.
- **prefix** (*str*, *optional*) – a prefix added to parameter and buffer names to compose the keys in *state\_dict*. Default: *''*.



- **keep\_vars** (*bool*, *optional*) – by default the Tensor s returned in the state dict are detached from autograd. If it's set to True, detaching will not be performed. Default: False.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(*\*args*, *\*\*kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(*device=None*, *dtype=None*, *non\_blocking=False*)

**to**(*dtype*, *non\_blocking=False*)

**to**(*tensor*, *non\_blocking=False*)

**to**(*memory\_format=torch.channels\_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** self

**Return type** Module

Examples:

```

>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (*type* or *string*) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *device* (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**zero\_grad**(*set\_to\_none: bool = False*) → None

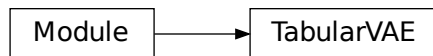
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** *set\_to\_none* (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

## synthcity.plugins.core.models.tabular\_vae module

```
class TabularVAE(X: pandas.core.frame.DataFrame, n_units_embedding: int, cond:
 Optional[Union[pandas.core.frame.DataFrame, pandas.core.series.Series, numpy.ndarray]] =
 None, lr: float = 0.0002, n_iter: int = 500, weight_decay: float = 0.001, batch_size: int = 64,
 random_state: int = 0, loss_strategy: str = 'standard', encoder_max_clusters: int = 20,
 decoder_n_layers_hidden: int = 2, decoder_n_units_hidden: int = 250, decoder_nonlin: str =
 'leaky_relu', decoder_nonlin_out_discrete: str = 'softmax', decoder_nonlin_out_continuous:
 str = 'tanh', decoder_batch_norm: bool = False, decoder_dropout: float = 0,
 decoder_residual: bool = True, encoder_n_layers_hidden: int = 3, encoder_n_units_hidden:
 int = 300, encoder_nonlin: str = 'leaky_relu', encoder_batch_norm: bool = False,
 encoder_dropout: float = 0.1, encoder_whitelist: list = [], device: Any = device(type='cpu'),
 robust_divergence_beta: int = 2, loss_factor: int = 1, dataloader_sampler:
 Optional[synthcity.utils.samplers.BaseSampler] = None, clipping_value: int = 1, n_iter_min:
 int = 100, n_iter_print: int = 10, patience: int = 20)
```

Bases: torch.nn.modules.module.Module



VAE for tabular data.

This class combines VAE and tabular encoder to form a generative model for tabular data.

#### Parameters

- **X** – `pd.DataFrame` Reference dataset, used for training the tabular encoder
- **cond** – Optional Optional conditional
- **decoder\_n\_layers\_hidden** – int Number of hidden layers in the decoder
- **decoder\_n\_units\_hidden** – int Number of hidden units in each layer of the decoder
- **decoder\_nonlin** – string, default ‘elu’ Nonlinearity to use in the decoder. Can be ‘elu’, ‘relu’, ‘selu’ or ‘leaky\_relu’.
- **decoder\_n\_iter** – int Maximum number of iterations in the decoder.
- **decoder\_batch\_norm** – bool Enable/disable batch norm for the decoder
- **decoder\_dropout** – float Dropout value. If 0, the dropout is not used.
- **decoder\_residual** – bool Use residuals for the decoder
- **encoder\_n\_layers\_hidden** – int Number of hidden layers in the encoder
- **encoder\_n\_units\_hidden** – int Number of hidden units in each layer of the encoder
- **encoder\_nonlin** – string, default ‘relu’ Nonlinearity to use in the encoder. Can be ‘elu’, ‘relu’, ‘selu’ or ‘leaky\_relu’.
- **encoder\_n\_iter** – int Maximum number of iterations in the encoder.
- **encoder\_batch\_norm** – bool Enable/disable batch norm for the encoder
- **encoder\_dropout** – float Dropout value for the encoder. If 0, the dropout is not used.
- **lr** – float learning rate for optimizer.
- **weight\_decay** – float l2 (ridge) penalty for the weights.
- **batch\_size** – int Batch size
- **random\_state** – int random\_state used
- **encoder\_max\_clusters** – int The max number of clusters to create for continuous columns when encoding
- **stopping** (# *early*) –
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.
- **n\_iter\_min** – int Minimum number of iterations to go through before starting early stopping

- **patience** – int Max number of iterations without any improvement before early stopping is triggered.

**T\_destination**

alias of `TypeVar('T_destination', bound=Dict[str, Any])`

**add\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

**Parameters**

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

**apply**(*fn: Callable[[torch.nn.modules.module.Module], None]*) → `torch.nn.modules.module.T`

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** *fn* (*Module* → None) – function to be applied to each submodule

**Returns** self

**Return type** Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
```

**bfloat16**() → `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `bfloat16` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**buffers**(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules.

Otherwise, yields only buffers that are direct members of this module.

**Yields** *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**children**() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module

**cpu**() → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** *Module*

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** *self*

**Return type** *Module*

**decode**(*X: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

**double**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to double datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** *Module*

**dump\_patches:** bool = False

**encode**(*X: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

**eval**() → torch.nn.modules.module.T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** self

**Return type** Module

**extra\_repr**() → str

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**fit**(*X: pandas.core.frame.DataFrame, cond: Optional[Union[pandas.core.frame.DataFrame, pandas.core.series.Series, numpy.ndarray]] = None, \*\*kwargs: Any*) → Any

**float**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to float datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**forward**(*count: int, cond: Optional[Union[pandas.core.frame.DataFrame, pandas.core.series.Series, numpy.ndarray]] = None*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**generate**(*count: int, cond: Optional[Union[pandas.core.frame.DataFrame, pandas.core.series.Series, numpy.ndarray]] = None, \*\*kwargs: Any*) → pandas.core.frame.DataFrame

**get\_buffer**(*target: str*) → torch.Tensor

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

**Parameters** **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by `target`

**Return type** torch.Tensor

**Raises `AttributeError`** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_encoder()** → *synthcity.plugins.core.models.tabular\_encoder.TabularEncoder*

**get\_extra\_state()** → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter(target: str)** → torch.nn.parameter.Parameter

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

**Parameters target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by `target`

**Return type** torch.nn.Parameter

**Raises `AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule(target: str)** → torch.nn.modules.module.Module

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)
```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters target** – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)



**Returns** The submodule referenced by `target`

**Return type** `torch.nn.Module`

**Raises `AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**`half()`**  $\rightarrow$  `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**`ipu(device: Optional[Union[int, torch.device]] = None)`**  $\rightarrow$  `torch.nn.modules.module.T`

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters `device` (`int`, `optional`)** – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**`load_state_dict(state_dict: Mapping[str, Any], strict: bool = True)`**

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

**Parameters**

- **`state_dict` (`dict`)** – a dict containing parameters and persistent buffers.
- **`strict` (`bool`, `optional`)** – whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `True`

**Returns**

- **`missing_keys`** is a list of `str` containing the missing keys
- **`unexpected_keys`** is a list of `str` containing the unexpected keys

**Return type** `NamedTuple` with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as `None` and its corresponding key exists in `state_dict`, `load_state_dict()` will raise a `RuntimeError`.

---

**`modules()`**  $\rightarrow$  `Iterator[torch.nn.modules.module.Module]`

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str, torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children**() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str, Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo: Optional[Set[torch.nn.modules.module.Module]] = None, prefix: str = "", remove\_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module

- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** *(str, Module)* – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, `l` will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters** **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_buffer**(name: str, tensor: Optional[torch.Tensor], persistent: bool = True) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

#### Parameters

- **name** (str) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (Tensor or None) – buffer to be registered. If `None`, then operations that run on buffers, such as `cuda`, are ignored. If `None`, the buffer is **not** included in the module's `state_dict`.
- **persistent** (bool) – whether the buffer is part of this module's `state_dict`.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_forward\_pre\_hook**(hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → `torch.utils.hooks.RemovableHandle`

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The module argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None  
Alias for [add\\_module\(\)](#).

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None  
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If *None*, then operations that run on parameters, such as [cuda](#), are ignored. If *None*, the parameter is **not** included in the module's [state\\_dict](#).

**requires\_grad\_**(*requires\_grad: bool = True*) → torch.nn.modules.module.T  
Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** **requires\_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: *True*.

**Returns** *self*

**Return type** Module

**set\_extra\_state**(*state: Any*)  
This function is called from [load\\_state\\_dict\(\)](#) to handle any extra state found within the *state\_dict*. Implement this function and a corresponding [get\\_extra\\_state\(\)](#) for your module if you need to store extra state within its *state\_dict*.

**Parameters** **state** (*dict*) – Extra state from the *state\_dict*

**share\_memory**() → torch.nn.modules.module.T  
See `torch.Tensor.share_memory_()`

**state\_dict**(\*args, *destination=None, prefix="", keep\_vars=False*)  
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to *None* are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument `destination` as it is not designed for end-users.

### Parameters

- **destination** (*dict*, *optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str*, *optional*) – a prefix added to parameter and buffer names to compose the keys in `state_dict`. Default: `''`.
- **keep\_vars** (*bool*, *optional*) – by default the `Tensor`s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** `dict`

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(*\*args*, *\*\*kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(*device=None*, *dtype=None*, *non\_blocking=False*)

**to**(*dtype*, *non\_blocking=False*)

**to**(*tensor*, *non\_blocking=False*)

**to**(*memory\_format=torch.channels\_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

- **memory\_format** (torch.memory\_format) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** self

**Return type** Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

**to\_empty**(\*, device: Union[str, torch.device]) → torch.nn.modules.module.T  
 Moves the parameters and buffers to the specified device without copying storage.

**Parameters** **device** (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module



**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** **mode** (*bool*) – whether to set training mode (True) or evaluation mode (False).

Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **dst\_type** (*type or string*) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**zero\_grad**(*set\_to\_none: bool = False*) → None

Sets gradients of all model parameters to zero. See similar function under torch.optim.Optimizer for more context.

**Parameters** **set\_to\_none** (*bool*) – instead of setting to zero, set the grads to None. See torch.optim.Optimizer.zero\_grad() for details.

**synthcity.plugins.core.models.tabular\_flows module**

```
class TabularFlows(X: pandas.core.frame.DataFrame, n_iter: int = 1000, n_layers_hidden: int = 5,
 n_units_hidden: int = 10, batch_size: int = 100, num_transform_blocks: int = 2, dropout:
 float = 0.25, batch_norm: bool = False, num_bins: int = 8, tail_bound: float = 3, lr: float =
 0.001, apply_unconditional_transform: bool = True, base_distribution: str =
 'standard_normal', linear_transform_type: str = 'permutation', base_transform_type: str =
 'rq-autoregressive', encoder_max_clusters: int = 20, encoder_whitelist: list = [], device:
 Any = device(type='cpu'), n_iter_min: int = 100, n_iter_print: int = 10, patience: int = 10,
 patience_metric: Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None)
```

Bases: torch.nn.modules.module.Module

Normalizing flow for tabular data.

This class combines normalizing flow and tabular encoder to form a generative model for tabular data.

**Parameters**

- **n\_iter** – int Number of flow steps
- **n\_layers\_hidden** – int Number of transformation layers
- **n\_units\_hidden** – int Number of hidden units for each layer
- **batch\_size** – int Size of batch used for training
- **num\_transform\_blocks** – int Number of blocks to use in coupling/autoregressive nets.
- **dropout** – float Dropout probability for coupling/autoregressive nets.
- **batch\_norm** – bool Whether to use batch norm in coupling/autoregressive nets.
- **num\_bins** – int Number of bins to use for piecewise transforms.
- **tail\_bound** – float Box is on  $[-bound, bound]^2$
- **lr** – float Learning rate for optimizer.
- **apply\_unconditional\_transform** – bool Whether to unconditionally transform ‘identity’ features in the coupling layer.
- **base\_distribution** – str Possible values: “standard\_normal”
- **linear\_transform\_type** – str Type of linear transform to use. Possible values:
  - lu : A linear transform where we parameterize the LU decomposition of the weights.
  - permutation: Permutes using a random, but fixed, permutation.
  - svd: A linear module using the SVD decomposition for the weight matrix.
- **base\_transform\_type** – str Type of transform to use between linear layers. Possible values:
  - **affine-coupling** [An affine coupling layer that scales and shifts part of the variables.] Ref: L. Dinh et al., “Density estimation using Real NVP”.
  - **quadratic-coupling** : Ref: Müller et al., “Neural Importance Sampling”.
  - **rq-coupling** [Rational Quadratic Coupling] Ref: Durkan et al, “Neural Spline Flows”.
  - **affine-autoregressive :Affine Autoregressive Transform** Ref: Durkan et al, “Neural Spline Flows”.

- **quadratic-autoregressive** [Quadratic Autoregressive Transform] Ref: Durkan et al, “Neural Spline Flows”.
- **rq-autoregressive** [Rational Quadratic Autoregressive Transform] Ref: Durkan et al, “Neural Spline Flows”.
- **stopping** (# *early*) –
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.
- **n\_iter\_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before early stopping is triggered.
- **patience\_metric** – WeightedMetrics Metric evaluator

**T\_destination**

alias of TypeVar(‘T\_destination’, bound=Dict[str, Any])

**add\_module**(name: str, module: Optional[torch.nn.modules.module.Module]) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

**Parameters**

- **name** (str) – name of the child module. The child module can be accessed from this module using the given name
- **module** (Module) – child module to be added to the module.

**apply**(fn: Callable[[torch.nn.modules.module.Module], None]) → torch.nn.modules.module.T

Applies fn recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** **fn** (Module → None) – function to be applied to each submodule

**Returns** self

**Return type** Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
```

(continues on next page)

(continued from previous page)

```
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

**Note:** This method modifies the module in-place.**Returns** self**Return type** Module**buffers**(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.**Yields** *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module**cpu()** → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

**Note:** This method modifies the module in-place.**Returns** self**Return type** Module**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

**Note:** This method modifies the module in-place.

**Parameters** `device` (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**decode**(*X: pandas.core.frame.DataFrame*) → `pandas.core.frame.DataFrame`

**double**() → `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `double` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**dump\_patches: bool = False**

**encode**(*X: pandas.core.frame.DataFrame*) → `pandas.core.frame.DataFrame`

**eval**() → `torch.nn.modules.module.T`

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** `self`

**Return type** `Module`

**extra\_repr**() → `str`

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**fit**(*X: pandas.core.frame.DataFrame*) → `Any`

**float**() → `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `float` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**forward**(*count: int*) → `torch.Tensor`

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**generate**(*count: int*) → `pandas.core.frame.DataFrame`

**get\_buffer**(*target: str*) → `torch.Tensor`

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

**Parameters** `target` – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by `target`

**Return type** `torch.Tensor`

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_encoder**() → *`synthcity.plugins.core.models.tabular_encoder.TabularEncoder`*

**get\_extra\_state**() → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding *`set_extra_state()`* for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → `torch.nn.parameter.Parameter`

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

**Parameters** `target` – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by `target`

**Return type** `torch.nn.Parameter`

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(*target: str*) → `torch.nn.modules.module.Module`

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 ...
)
)
)
```

(continues on next page)

(continued from previous page)

```

 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)

```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by `target`

**Return type** `torch.nn.Module`

**Raises** **`AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**`half()`**  $\rightarrow$  `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**`ipu(device: Optional[Union[int, torch.device]] = None)`**  $\rightarrow$  `torch.nn.modules.module.T`

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device (int, optional)` – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**`load_state_dict(state_dict: Mapping[str, Any], strict: bool = True)`**

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

**Parameters**

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `True`

**Returns**

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** `NamedTuple` with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as `None` and its corresponding key exists in `state_dict`, `load_state_dict()` will raise a `RuntimeError`.

---

**modules()** → `Iterator[torch.nn.modules.module.Module]`

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, `l` will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix: str = "", recurse: bool = True*) → `Iterator[Tuple[str, torch.Tensor]]`

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

**Parameters**

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if `True`, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str*, *torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```



**named\_children()** → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str*, *Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo*: Optional[Set[torch.nn.modules.module.Module]] = None, *prefix*: str = "", *remove\_duplicate*: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str*, *Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, `l` will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str*, *Parameter*) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(recurse: bool = True) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters** **recurse** (bool) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_buffer**(name: str, tensor: Optional[torch.Tensor], persistent: bool = True) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to False. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

#### Parameters

- **name** (str) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (Tensor or None) – buffer to be registered. If None, then operations that run on buffers, such as `cuda`, are ignored. If None, the buffer is **not** included in the module's `state_dict`.
- **persistent** (bool) – whether the buffer is part of this module's `state_dict`.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_forward\_pre\_hook**(hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_full\_backward\_hook**(hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*)  $\rightarrow$  None

Alias for `add_module()`.

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*)  $\rightarrow$  None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad\_**(*requires\_grad: bool = True*)  $\rightarrow$  `torch.nn.modules.module.T`

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** **requires\_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** `self`

**Return type** `Module`

**set\_extra\_state**(*state: Any*)

This function is called from `load_state_dict()` to handle any extra state found within the `state_dict`. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its `state_dict`.

**Parameters** **state** (*dict*) – Extra state from the `state_dict`

**share\_memory()** → torch.nn.modules.module.T

See torch.Tensor.share\_memory\_()

**state\_dict(\*args, destination=None, prefix="", keep\_vars=False)**

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to None are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently state\_dict() also accepts positional arguments for destination, prefix and keep\_vars in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument destination as it is not designed for end-users.

### Parameters

- **destination** (*dict*, *optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an OrderedDict will be created and returned. Default: None.
- **prefix** (*str*, *optional*) – a prefix added to parameter and buffer names to compose the keys in state\_dict. Default: ''.
- **keep\_vars** (*bool*, *optional*) – by default the Tensor s returned in the state dict are detached from autograd. If it's set to True, detaching will not be performed. Default: False.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to(\*args, \*\*kwargs)**

Moves and/or casts the parameters and buffers.

This can be called as

**to**(device=None, dtype=None, non\_blocking=False)

**to**(dtype, non\_blocking=False)

**to**(tensor, non\_blocking=False)

**to**(memory\_format=torch.channels\_last)

Its signature is similar to torch.Tensor.to(), but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to dtype (if given). The integral parameters and buffers will be moved device, if that is given, but with dtypes unchanged.

When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** `self`

**Return type** `Module`

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)
```

(continues on next page)

(continued from previous page)

```

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (type or string) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *device* (int, optional) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**zero\_grad**(*set\_to\_none: bool = False*) → None

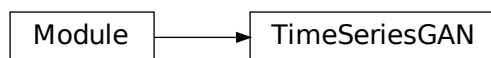
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** **set\_to\_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

### `synthcity.plugins.core.models.ts_gan` module

```
class TimeSeriesGAN(n_static_units: int, n_static_units_latent: int, n_temporal_units: int, n_temporal_window: int, n_temporal_units_latent: int, n_units_conditional: int = 0, generator_n_layers_hidden: int = 1, generator_n_units_hidden: int = 250, generator_nonlin: str = 'leaky_relu', generator_static_nonlin_out: Optional[List[Tuple[str, int]]] = None, generator_temporal_nonlin_out: Optional[List[Tuple[str, int]]] = None, generator_n_iter: int = 1000, generator_batch_norm: bool = False, generator_dropout: float = 0, generator_loss: Optional[Callable] = None, generator_lr: float = 0.0002, generator_weight_decay: float = 0.001, generator_residual: bool = True, discriminator_n_layers_hidden: int = 1, discriminator_n_units_hidden: int = 300, discriminator_nonlin: str = 'leaky_relu', discriminator_n_iter: int = 1, discriminator_batch_norm: bool = False, discriminator_dropout: float = 0.1, discriminator_loss: Optional[Callable] = None, discriminator_lr: float = 0.0002, discriminator_weight_decay: float = 0.001, batch_size: int = 64, n_iter_print: int = 10, random_state: int = 0, clipping_value: int = 1, gamma_penalty: float = 1, moments_penalty: float = 100, embedding_penalty: float = 10, dataloader_sampler: Optional[torch.utils.data.sampler.Sampler] = None, mode: str = 'RNN', device: Any = device(type='cpu'), use_horizon_condition: bool = True)
```

Bases: `torch.nn.modules.module.Module`



Basic TimeSeriesGAN implementation.

#### Parameters

- **n\_static\_units** – int, Number of units for the static features
- **n\_static\_units\_latent** – int, Number of latent units for the static features
- **n\_temporal\_units** – int, Number of units for the temporal features
- **n\_temporal\_window** – int, Number of temporal observations for each subject
- **n\_temporal\_units\_latent** – int, Number of temporal latent units
- **n\_units\_conditional** – int = 0, Number of conditional units
- **generator\_n\_layers\_hidden** – int. Default: 1 Number of hidden layers in the generator



- **generator\_n\_units\_hidden** – int. Default: 250 Number of hidden units in each layer of the Generator
- **generator\_nonlin** – string, default ‘elu’ Nonlinearity to use in the generator. Can be ‘elu’, ‘relu’, ‘selu’ or ‘leaky\_relu’.
- **generator\_n\_iter** – int. Default: 1000 Maximum number of iterations in the Generator.
- **generator\_batch\_norm** – bool. Default: False Enable/disable batch norm for the generator
- **generator\_dropout** – float. Default: 0 Generator Dropout value. If 0, the dropout is not used.
- **generator\_residual** – bool. Default: True Use residuals for the generator
- **generator\_lr** – float. Default: 2e-4 Generator learning rate.
- **generator\_weight\_decay** – float. Default = 1e-3 l2 (ridge) penalty for the generator weights.
- **discriminator\_n\_layers\_hidden** – int. Default = 1 Number of hidden layers in the discriminator
- **discriminator\_n\_units\_hidden** – int. Default = 300 Number of hidden units in each layer of the discriminator
- **discriminator\_nonlin** – string, default = ‘relu’ Nonlinearity to use in the discriminator. Can be ‘elu’, ‘relu’, ‘selu’ or ‘leaky\_relu’.
- **discriminator\_n\_iter** – int Maximum number of iterations in the discriminator.
- **discriminator\_batch\_norm** – bool. Default: False Enable/disable batch norm for the discriminator
- **discriminator\_dropout** – float. Default = 0.1 Dropout value for the discriminator. If 0, the dropout is not used.
- **discriminator\_lr** – float. Default = 2e-4 learning rate for discriminator optimizer.
- **discriminator\_weight\_decay** – float. Default = 1e-3 l2 (ridge) penalty for the discriminator weights.
- **batch\_size** – int. Default = 64 Batch size
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.
- **random\_state** – int random\_state used
- **clipping\_value** – int, default = 0 Gradients clipping value. Zero disables the feature
- **gamma\_penalty** – float. Default = 1. Latent representation penalty
- **moments\_penalty** – float. Default = 100. Generator Moments(var and mean) penalty
- **embedding\_penalty** – float. Default = 10 Embedding representation penalty
- **dataloader\_sampler** – Optional[sampler.Sampler] = None Optional data sampler
- **mode** – str = “RNN” Core neural net architecture. Available models:
  - “LSTM”
  - “GRU”

- "RNN"
  - "Transformer"
  - "MLSTM\_FCN"
  - "TCN"
  - "InceptionTime"
  - "InceptionTimePlus"
  - "XceptionTime"
  - "ResCNN"
  - "OmniScaleCNN"
  - "XCM"
- **device** – The device used by PyTorch. `cpu/cuda`
  - **use\_horizon\_condition** – bool. Default = True Whether to condition the covariate generation on the observation times or not.

**T\_destination**

alias of `TypeVar('T_destination', bound=Dict[str, Any])`

**add\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

**Parameters**

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

**apply**(*fn: Callable[[torch.nn.modules.module.Module], None]*) → `torch.nn.modules.module.T`

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** *fn* (*Module* → None) – function to be applied to each submodule

**Returns** self

**Return type** Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
```

(continues on next page)

(continued from previous page)

```

Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)

```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**buffers**(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** *torch.Tensor* – module buffer

Example:

```

>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module

**cpu()** → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** *self*

**Return type** *Module*

**dataloader**(*static\_data: torch.Tensor, temporal\_data: torch.Tensor, observation\_times: torch.Tensor, cond: Optional[torch.Tensor] = None*) → *torch.utils.data.dataloader.DataLoader*

**double**() → *torch.nn.modules.module.T*  
Casts all floating point parameters and buffers to *double* datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** *Module*

**dump\_patches: bool = False**

**eval**() → *torch.nn.modules.module.T*  
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. *Dropout*, *BatchNorm*, etc.

This is equivalent with *self.train(False)*.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

**Returns** *self*

**Return type** *Module*

**extra\_repr**() → *str*  
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**fit**(*static\_data: numpy.ndarray, temporal\_data: numpy.ndarray, observation\_times: numpy.ndarray, cond: Optional[numpy.ndarray] = None*) → *synthcity.plugins.core.models.ts\_gan.TimeSeriesGAN*

**float**() → *torch.nn.modules.module.T*  
Casts all floating point parameters and buffers to *float* datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** *Module*

**forward**(*count: int, cond: Optional[torch.Tensor] = None, static\_data: Optional[torch.Tensor] = None, observation\_times: Optional[torch.Tensor] = None*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**generate**(*count: int, cond: Optional[numpy.ndarray] = None, static\_data: Optional[numpy.ndarray] = None, observation\_times: Optional[numpy.ndarray] = None*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

**get\_buffer**(*target: str*) → torch.Tensor

Returns the buffer given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by *target*

**Return type** torch.Tensor

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by *target*

**Return type** torch.nn.Parameter

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by *target* if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)
```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by `target`

**Return type** `torch.nn.Module`

**Raises** **`AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**`half()`** → `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**`ipu(device: Optional[Union[int, torch.device]] = None)`** → `torch.nn.modules.module.T`

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (`int`, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**load\_state\_dict**(*state\_dict*: Mapping[str, Any], *strict*: bool = True)

Copies parameters and buffers from *state\_dict* into this module and its descendants. If *strict* is True, then the keys of *state\_dict* must exactly match the keys returned by this module's `state_dict()` function.

#### Parameters

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in *state\_dict* match the keys returned by this module's `state_dict()` function. Default: True

#### Returns

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** NamedTuple with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as None and its corresponding key exists in *state\_dict*, `load_state_dict()` will raise a `RuntimeError`.

---

**modules**() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str*, *torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children()** → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str*, *Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo*: Optional[Set[torch.nn.modules.module.Module]] = None, *prefix*: str = "", *remove\_duplicate*: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str*, *Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.



- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str*, *Parameter*) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_buffer**(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm’s `running_mean` is not a parameter, but is part of the module’s state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to False. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module’s `state_dict`.

Buffers can be accessed as attributes using given names.

**Parameters**

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If None, then operations that run on buffers, such as `cuda`, are ignored. If None, the buffer is **not** included in the module’s `state_dict`.

- **persistent** (*bool*) – whether the buffer is part of this module’s *state\_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → *torch.utils.hooks.RemovableHandle*

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The *grad\_input* and *grad\_output* are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of *grad\_input* in subsequent computations. *grad\_input* will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in *grad\_input* and *grad\_output* will be *None* for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module’s forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Alias for `add_module()`.

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad**(*requires\_grad: bool = True*) → `torch.nn.modules.module.T`

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** `requires_grad` (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** `self`

**Return type** `Module`

**set\_extra\_state**(*state: Any*)

This function is called from `load_state_dict()` to handle any extra state found within the *state\_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state\_dict*.

**Parameters** *state (dict)* – Extra state from the *state\_dict*

**share\_memory**() → torch.nn.modules.module.T

See torch.Tensor.share\_memory\_()

**state\_dict**(\*args, destination=None, prefix="", keep\_vars=False)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to None are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently `state_dict()` also accepts positional arguments for *destination*, *prefix* and *keep\_vars* in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument *destination* as it is not designed for end-users.

### Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in *state\_dict*. Default: `''`.
- **keep\_vars** (*bool, optional*) – by default the `Tensor`s returned in the *state dict* are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(\*args, \*\*kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(*device=None, dtype=None, non\_blocking=False*)

**to**(*dtype, non\_blocking=False*)

`to(tensor, non_blocking=False)`

`to(memory_format=torch.channels_last)`

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** `self`

**Return type** `Module`

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```

Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (type or string) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**zero\_grad**(*set\_to\_none: bool = False*) → `None`

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** `set_to_none` (*bool*) – instead of setting to zero, set the grads to `None`. See `torch.optim.Optimizer.zero_grad()` for details.

### `synthcity.plugins.core.models.ts_vae` module

**class** `LatentODE`(*n\_units\_embedding: int*, *n\_layers\_hidden: int = 1*, *n\_units\_hidden: int = 100*, *nonlin: str = 'relu'*, *random\_state: int = 0*, *dropout: float = 0.1*, *batch\_norm: bool = True*, *residual: bool = False*, *device: Any = device(type='cpu')*)

Bases: `torch.nn.modules.module.Module`

**T\_destination**

alias of `TypeVar('T_destination', bound=Dict[str, Any])`

**add\_module**(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → `None`

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

**Parameters**

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

**apply**(*fn: Callable[[torch.nn.modules.module.Module], None]*) → `torch.nn.modules.module.T`

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as `self`. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** `fn` (*Module -> None*) – function to be applied to each submodule

**Returns** `self`

**Return type** `Module`

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
```

(continues on next page)

(continued from previous page)

```

Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)

```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**buffers**(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** *torch.Tensor* – module buffer

Example:

```

>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module

**cpu()** → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module



**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** *self*

**Return type** Module

**double**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to double datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** Module

**dump\_patches: bool = False**

**eval**() → torch.nn.modules.module.T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** *self*

**Return type** Module

**extra\_repr**() → str

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**float**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to float datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** Module

**forward**(*observation\_times: torch.Tensor, temporal: torch.Tensor*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**get\_buffer**(*target: str*) → torch.Tensor

Returns the buffer given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by *target*

**Return type** torch.Tensor

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by *target*

**Return type** torch.nn.Parameter

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by *target* if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 ...
)
)
```

(continues on next page)

(continued from previous page)

```

 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)

```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by `target`

**Return type** `torch.nn.Module`

**Raises** **`AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**`half()`**  $\rightarrow$  `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**`ipu(device: Optional[Union[int, torch.device]] = None)`**  $\rightarrow$  `torch.nn.modules.module.T`

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (`int`, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**load\_state\_dict**(*state\_dict*: Mapping[str, Any], *strict*: bool = True)

Copies parameters and buffers from *state\_dict* into this module and its descendants. If *strict* is True, then the keys of *state\_dict* must exactly match the keys returned by this module's `state_dict()` function.

#### Parameters

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in *state\_dict* match the keys returned by this module's `state_dict()` function. Default: True

#### Returns

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** NamedTuple with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as None and its corresponding key exists in *state\_dict*, `load_state_dict()` will raise a `RuntimeError`.

---

**modules**() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str*, *torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children()** → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str*, *Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo*: Optional[Set[torch.nn.modules.module.Module]] = None, *prefix*: str = "", *remove\_duplicate*: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str*, *Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str*, *Parameter*) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → `Iterator[torch.nn.parameter.Parameter]`

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → `torch.utils.hooks.RemovableHandle`

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_buffer**(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → `None`

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm’s `running_mean` is not a parameter, but is part of the module’s state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to False. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module’s `state_dict`.

Buffers can be accessed as attributes using given names.

**Parameters**

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If None, then operations that run on buffers, such as `cuda`, are ignored. If None, the buffer is **not** included in the module’s `state_dict`.

- **persistent** (*bool*) – whether the buffer is part of this module’s *state\_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → *torch.utils.hooks.RemovableHandle*

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The *grad\_input* and *grad\_output* are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of *grad\_input* in subsequent computations. *grad\_input* will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in *grad\_input* and *grad\_output* will be *None* for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module’s forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Alias for `add_module()`.

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad\_**(*requires\_grad: bool = True*) → `torch.nn.modules.module.T`

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** `requires_grad` (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** `self`

**Return type** `Module`



**set\_extra\_state**(*state: Any*)

This function is called from `load_state_dict()` to handle any extra state found within the *state\_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state\_dict*.

**Parameters** *state (dict)* – Extra state from the *state\_dict*

**share\_memory**() → torch.nn.modules.module.T

See `torch.Tensor.share_memory_()`

**state\_dict**(\*args, destination=None, prefix="", keep\_vars=False)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently `state_dict()` also accepts positional arguments for *destination*, *prefix* and *keep\_vars* in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument *destination* as it is not designed for end-users.

### Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in *state\_dict*. Default: `''`.
- **keep\_vars** (*bool, optional*) – by default the `Tensor`s returned in the *state dict* are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(\*args, \*\*kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(*device=None, dtype=None, non\_blocking=False*)

**to**(*dtype, non\_blocking=False*)

```
to(tensor, non_blocking=False)
```

```
to(memory_format=torch.channels_last)
```

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** `self`

**Return type** `Module`

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```

Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (type or string) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**zero\_grad**(*set\_to\_none: bool = False*) → `None`

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** `set_to_none` (*bool*) – instead of setting to zero, set the grads to `None`. See `torch.optim.Optimizer.zero_grad()` for details.

**class TimeSeriesDecoder**(*n\_static\_units\_embedding: int, n\_static\_units\_out: int, n\_temporal\_units\_embedding: int, n\_temporal\_units\_out: int, n\_temporal\_window: int, n\_layers\_hidden: int = 1, n\_units\_hidden: int = 100, nonlin: str = 'relu', static\_nonlin\_out: Optional[List[Tuple[str, int]]] = None, temporal\_nonlin\_out: Optional[List[Tuple[str, int]]] = None, random\_state: int = 0, dropout: float = 0.1, mode: str = 'LSTM', batch\_norm: bool = False, residual: bool = False, device: Any = device(type='cpu')*)

Bases: `torch.nn.modules.module.Module`

**T\_destination**

alias of `TypeVar('T_destination', bound=Dict[str, Any])`

**add\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → `None`

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

**Parameters**

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

**apply**(*fn: Callable[[torch.nn.modules.module.Module], None]*) → `torch.nn.modules.module.T`

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as `self`. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** `fn` (*Module -> None*) – function to be applied to each submodule

**Returns** `self`

**Return type** `Module`

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
```

(continues on next page)

(continued from previous page)

```
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**buffers**(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module

**cpu()** → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**double**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to double datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**dump\_patches: bool = False**

**eval**() → torch.nn.modules.module.T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** self

**Return type** Module

**extra\_repr**() → str

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**float**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to float datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**forward**(*static\_embs: torch.Tensor, temporal\_embs: torch.Tensor, horizon\_embs: torch.Tensor*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**get\_buffer**(*target: str*) → torch.Tensor

Returns the buffer given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by *target*

**Return type** torch.Tensor

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by *target*

**Return type** torch.nn.Parameter

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by *target* if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```

A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)

```

(The diagram shows an `nn.Module` A. A has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by `target`

**Return type** `torch.nn.Module`

**Raises** **`AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**half()** → `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**ipu(device: Optional[Union[int, torch.device]] = None)** → `torch.nn.modules.module.T`

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device (int, optional)` – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`



**load\_state\_dict**(*state\_dict*: Mapping[str, Any], *strict*: bool = True)

Copies parameters and buffers from *state\_dict* into this module and its descendants. If *strict* is True, then the keys of *state\_dict* must exactly match the keys returned by this module's `state_dict()` function.

#### Parameters

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in *state\_dict* match the keys returned by this module's `state_dict()` function. Default: True

#### Returns

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** NamedTuple with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as None and its corresponding key exists in *state\_dict*, `load_state_dict()` will raise a `RuntimeError`.

---

**modules**() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str*, *torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children()** → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str*, *Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo*: Optional[Set[torch.nn.modules.module.Module]] = None, *prefix*: str = "", *remove\_duplicate*: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str*, *Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str, Parameter*) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_buffer**(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm’s `running_mean` is not a parameter, but is part of the module’s state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to False. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module’s `state_dict`.

Buffers can be accessed as attributes using given names.

**Parameters**

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If None, then operations that run on buffers, such as `cuda`, are ignored. If None, the buffer is **not** included in the module’s `state_dict`.

- **persistent** (*bool*) – whether the buffer is part of this module’s *state\_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → *torch.utils.hooks.RemovableHandle*

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The *grad\_input* and *grad\_output* are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of *grad\_input* in subsequent computations. *grad\_input* will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in *grad\_input* and *grad\_output* will be *None* for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module’s forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Alias for `add_module()`.

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad**(*requires\_grad: bool = True*) → `torch.nn.modules.module.T`

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** `requires_grad` (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** `self`

**Return type** `Module`

**set\_extra\_state**(*state: Any*)

This function is called from `load_state_dict()` to handle any extra state found within the *state\_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state\_dict*.

**Parameters** *state (dict)* – Extra state from the *state\_dict*

**share\_memory**() → torch.nn.modules.module.T

See torch.Tensor.share\_memory\_()

**state\_dict**(\*args, destination=None, prefix="", keep\_vars=False)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to None are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently `state_dict()` also accepts positional arguments for *destination*, *prefix* and *keep\_vars* in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument *destination* as it is not designed for end-users.

### Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: None.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in *state\_dict*. Default: ''.
- **keep\_vars** (*bool, optional*) – by default the `Tensor`s returned in the *state dict* are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(\*args, \*\*kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(*device=None, dtype=None, non\_blocking=False*)

**to**(*dtype, non\_blocking=False*)

`to(tensor, non_blocking=False)`

`to(memory_format=torch.channels_last)`

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** `self`

**Return type** `Module`

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```

Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (type or string) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.



---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**zero\_grad**(*set\_to\_none: bool = False*) → `None`

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** `set_to_none` (*bool*) – instead of setting to zero, set the grads to `None`. See `torch.optim.Optimizer.zero_grad()` for details.

**class TimeSeriesEncoder**(*n\_static\_units: int, n\_static\_units\_embedding: int, n\_temporal\_units: int, n\_temporal\_window: int, n\_temporal\_units\_embedding: int, n\_units\_hidden: int = 200, n\_layers\_hidden: int = 2, nonlin: str = 'relu', batch\_norm: bool = False, dropout: float = 0.1, random\_state: int = 0, mode: str = 'LSTM', device: Any = device(type='cpu')*)

Bases: `torch.nn.modules.module.Module`

**T\_destination**

alias of `TypeVar('T_destination', bound=Dict[str, Any])`

**add\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → `None`

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

**Parameters**

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

**apply**(*fn: Callable[[torch.nn.modules.module.Module], None]*) → `torch.nn.modules.module.T`

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as `self`. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** `fn` (*Module -> None*) – function to be applied to each submodule

**Returns** `self`

**Return type** `Module`

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
```

(continues on next page)

(continued from previous page)

```

Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)

```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**buffers**(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** *torch.Tensor* – module buffer

Example:

```

>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module

**cpu()** → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**double()** → `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `double` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**dump\_patches:** `bool = False`

**eval()** → `torch.nn.modules.module.T`

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** `self`

**Return type** `Module`

**extra\_repr()** → `str`

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**float()** → `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `float` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**forward**(*static: torch.Tensor, temporal: torch.Tensor, observation\_times: torch.Tensor*) →

`Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]`

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**get\_buffer**(*target: str*) → torch.Tensor

Returns the buffer given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by *target*

**Return type** torch.Tensor

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by *target*

**Return type** torch.nn.Parameter

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by *target* if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
)
)
```

(continues on next page)

(continued from previous page)

```

 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)

```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by `target`

**Return type** `torch.nn.Module`

**Raises** **`AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**`half()`**  $\rightarrow$  `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**`ipu(device: Optional[Union[int, torch.device]] = None)`**  $\rightarrow$  `torch.nn.modules.module.T`

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device (int, optional)` – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**`load_state_dict(state_dict: Mapping[str, Any], strict: bool = True)`**

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

**Parameters**

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `True`

**Returns**

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** `NamedTuple` with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as `None` and its corresponding key exists in `state_dict`, `load_state_dict()` will raise a `RuntimeError`.

---

**modules()** → `Iterator[torch.nn.modules.module.Module]`

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, `l` will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers** (*prefix: str = "", recurse: bool = True*) → `Iterator[Tuple[str, torch.Tensor]]`

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

**Parameters**

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if `True`, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str, torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children**() → `Iterator[Tuple[str, torch.nn.modules.module.Module]]`

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str*, *Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo*: Optional[Set[torch.nn.modules.module.Module]] = None, *prefix*: str = "", *remove\_duplicate*: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str*, *Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str*, *Parameter*) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
```

(continues on next page)

(continued from previous page)

```
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(recurse: bool = True) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (bool) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_buffer**(name: str, tensor: Optional[torch.Tensor], persistent: bool = True) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to False. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

#### Parameters

- **name** (str) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (Tensor or None) – buffer to be registered. If None, then operations that run on buffers, such as `cuda`, are ignored. If None, the buffer is **not** included in the module's `state_dict`.
- **persistent** (bool) – whether the buffer is part of this module's `state_dict`.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```



**register\_forward\_hook**(hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_forward\_pre\_hook**(hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_full\_backward\_hook**(hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*)  $\rightarrow$  None

Alias for `add_module()`.

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*)  $\rightarrow$  None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad\_**(*requires\_grad: bool = True*)  $\rightarrow$  `torch.nn.modules.module.T`

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** **requires\_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** `self`

**Return type** `Module`

**set\_extra\_state**(*state: Any*)

This function is called from `load_state_dict()` to handle any extra state found within the `state_dict`. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its `state_dict`.

**Parameters** **state** (*dict*) – Extra state from the `state_dict`

**share\_memory()** → torch.nn.modules.module.T

See torch.Tensor.share\_memory\_()

**state\_dict**(\*args, destination=None, prefix="", keep\_vars=False)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to None are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently state\_dict() also accepts positional arguments for destination, prefix and keep\_vars in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument destination as it is not designed for end-users.

### Parameters

- **destination** (*dict*, *optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an OrderedDict will be created and returned. Default: None.
- **prefix** (*str*, *optional*) – a prefix added to parameter and buffer names to compose the keys in state\_dict. Default: ''.
- **keep\_vars** (*bool*, *optional*) – by default the Tensor s returned in the state dict are detached from autograd. If it's set to True, detaching will not be performed. Default: False.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(\*args, \*\*kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(device=None, dtype=None, non\_blocking=False)

**to**(dtype, non\_blocking=False)

**to**(tensor, non\_blocking=False)

**to**(memory\_format=torch.channels\_last)

Its signature is similar to torch.Tensor.to(), but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to dtype (if given). The integral parameters and buffers will be moved device, if that is given, but with dtypes unchanged.

When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** `self`

**Return type** `Module`

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)
```

(continues on next page)

(continued from previous page)

```

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (type or string) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *device* (int, optional) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

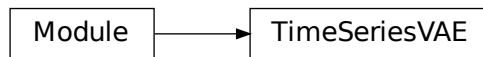
**zero\_grad**(*set\_to\_none: bool = False*) → None

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** **set\_to\_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

```
class TimeSeriesVAE(n_static_units: int, n_static_units_embedding: int, n_temporal_units: int,
 n_temporal_window: int, n_temporal_units_embedding: int, batch_size: int = 100, n_iter:
 int = 500, n_iter_print: int = 10, random_state: int = 0, clipping_value: int = 1, lr: float =
 0.0002, weight_decay: float = 0.001, loss_factor: int = 2, decoder_n_layers_hidden: int =
 2, decoder_n_units_hidden: int = 250, decoder_nonlin: str = 'leaky_relu',
 decoder_static_nonlin_out: Optional[List[Tuple[str, int]]] = None,
 decoder_temporal_nonlin_out: Optional[List[Tuple[str, int]]] = None,
 decoder_batch_norm: bool = False, decoder_dropout: float = 0, decoder_residual: bool
 = True, decoder_mode: str = 'LSTM', encoder_n_layers_hidden: int = 3,
 encoder_n_units_hidden: int = 300, encoder_nonlin: str = 'leaky_relu',
 encoder_batch_norm: bool = False, encoder_dropout: float = 0.1, encoder_mode: str =
 'LSTM', device: Any = device(type='cpu'))
```

Bases: `torch.nn.modules.module.Module`



Basic Time-Series Variational AutoEncoder

#### Parameters

- **n\_static\_units** – int Number of input static units.
- **n\_static\_units\_embedding** – int. Number of latent static units.
- **n\_temporal\_units** – int. Number of input temporal units.
- **n\_temporal\_window** – int The length of time series sequences.
- **n\_temporal\_units\_embedding** – int. Number of latent temporal units.
- **batch\_size** – int. Default = 100 Batch size.
- **n\_iter** – int. Default = 500 Number of epochs
- **n\_iter\_print** – int = 10. Frequency of printing the training loss.
- **random\_state** – int = 0 Random seed
- **clipping\_value** – int = 1 Gradients clipping value. Zero disables the feature
- **lr** – float = 2e-4, Learning rate
- **weight\_decay** – float = 1e-3, l2 (ridge) penalty for the weights.
- **loss\_factor** – int. Default = 2 Reconstruction loss weight.

- **decoder\_n\_layers\_hidden** – int. Default = 2 Number of hidden layer in the decoder
- **decoder\_n\_units\_hidden** – int. Decoder = 250. Number of hidden units in the decoder
- **decoder\_nonlin** – str. Decoder = “leaky\_relu” Activation for the hidden layers. Can be relu, elu, leaky\_relu, selu.
- **decoder\_static\_nonlin\_out** – Optional[List[Tuple[str, int]]] = None (Optional) Activations to use in the output layer of the decoder for static features.
- **decoder\_temporal\_nonlin\_out** – Optional[List[Tuple[str, int]]] = None, (Optional) Activations to use in the output layer of the decoder for temporal features.
- **decoder\_batch\_norm** – bool. Default = False Decoder batch norm
- **decoder\_dropout** – float. Default = 0 Decoder dropout
- **decoder\_residual** – bool. Default = True Use residual connections in the decoder
- **decoder\_mode** – str. Default = “LSTM” Core neural net architecture for the decoder.  
Available models:
  - “LSTM”
  - “GRU”
  - “RNN”
  - “Transformer”
  - “MLSTM\_FCN”
  - “TCN”
  - “InceptionTime”
  - “InceptionTimePlus”
  - “XceptionTime”
  - “ResCNN”
  - “OmniScaleCNN”
  - “XCM”
- **encoder\_n\_layers\_hidden** – int. Default = 3 Number of hidden layers in the encoder
- **300** (*encoder\_n\_units\_hidden. Default int* =) – Number of hidden units in the encoder
- **encoder\_nonlin** – str. Default = “leaky\_relu” Activations for the hidden layers in the encoder.
- **encoder\_batch\_norm** – bool. Default = False, Encoder batch norm.
- **encoder\_dropout** – float. Default = 0.1 Encoder dropout.
- **encoder\_mode** – str. Default = “LSTM” Core neural net architecture for the encoder.  
Available models:
  - “LSTM”
  - “GRU”
  - “RNN”
  - “Transformer”

- "MLSTM\_FCN"
- "TCN"
- "InceptionTime"
- "InceptionTimePlus"
- "XceptionTime"
- "ResCNN"
- "OmniScaleCNN"
- "XCM"
- "Transformer"

- **device** – PyTorch device: cpu/cuda.

### **T\_destination**

alias of TypeVar('T\_destination', bound=Dict[str, Any])

**add\_module**(name: str, module: Optional[torch.nn.modules.module.Module]) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

### **Parameters**

- **name** (str) – name of the child module. The child module can be accessed from this module using the given name
- **module** (Module) – child module to be added to the module.

**apply**(fn: Callable[[torch.nn.modules.module.Module], None]) → torch.nn.modules.module.T

Applies fn recursively to every submodule (as returned by .children()) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

**Parameters** fn (Module → None) – function to be applied to each submodule

**Returns** self

**Return type** Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
```

(continues on next page)



(continued from previous page)

```
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**buffers**(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module

**cpu()** → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**double()** → `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `double` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**dump\_patches:** `bool = False`

**eval()** → `torch.nn.modules.module.T`

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** `self`

**Return type** `Module`

**extra\_repr()** → `str`

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**fit**(*static: numpy.ndarray, temporal: numpy.ndarray, observation\_times: numpy.ndarray*) → `Any`

**float()** → `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `float` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**forward**(\**input: Any*) → `None`

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**generate**(*samples: int*) → `numpy.ndarray`

**get\_buffer**(*target: str*) → `torch.Tensor`

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

**Parameters** `target` – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by `target`

**Return type** `torch.Tensor`

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → `torch.nn.parameter.Parameter`

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

**Parameters** `target` – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by `target`

**Return type** `torch.nn.Parameter`

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(*target: str*) → `torch.nn.modules.module.Module`

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
)
```

(continues on next page)

(continued from previous page)

```

 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)

```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by `target`

**Return type** `torch.nn.Module`

**Raises** **`AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**`half()`**  $\rightarrow$  `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**`ipu(device: Optional[Union[int, torch.device]] = None)`**  $\rightarrow$  `torch.nn.modules.module.T`

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device (int, optional)` – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**`load_state_dict(state_dict: Mapping[str, Any], strict: bool = True)`**

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

**Parameters**

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `True`

**Returns**

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** `NamedTuple` with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as `None` and its corresponding key exists in `state_dict`, `load_state_dict()` will raise a `RuntimeError`.

---

**modules()** → `Iterator[torch.nn.modules.module.Module]`

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, `l` will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers** (*prefix: str = "", recurse: bool = True*) → `Iterator[Tuple[str, torch.Tensor]]`

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

**Parameters**

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if `True`, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str, torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children**() → `Iterator[Tuple[str, torch.nn.modules.module.Module]]`

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str*, *Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo*: Optional[Set[torch.nn.modules.module.Module]] = None, *prefix*: str = "", *remove\_duplicate*: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str*, *Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str*, *Parameter*) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
```

(continues on next page)

(continued from previous page)

```
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(recurse: bool = True) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (bool) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_buffer**(name: str, tensor: Optional[torch.Tensor], persistent: bool = True) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to False. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

#### Parameters

- **name** (str) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (Tensor or None) – buffer to be registered. If None, then operations that run on buffers, such as `cuda`, are ignored. If None, the buffer is **not** included in the module's `state_dict`.
- **persistent** (bool) – whether the buffer is part of this module's `state_dict`.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_forward\_pre\_hook**(hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_full\_backward\_hook**(hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`



**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Alias for `add_module()`.

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad\_**(*requires\_grad: bool = True*) → `torch.nn.modules.module.T`

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** **requires\_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** `self`

**Return type** `Module`

**set\_extra\_state**(*state: Any*)

This function is called from `load_state_dict()` to handle any extra state found within the `state_dict`. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its `state_dict`.

**Parameters** **state** (*dict*) – Extra state from the `state_dict`

**share\_memory()** → torch.nn.modules.module.T

See torch.Tensor.share\_memory\_()

**state\_dict(\*args, destination=None, prefix="", keep\_vars=False)**

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to None are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently state\_dict() also accepts positional arguments for destination, prefix and keep\_vars in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument destination as it is not designed for end-users.

### Parameters

- **destination** (*dict*, *optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an OrderedDict will be created and returned. Default: None.
- **prefix** (*str*, *optional*) – a prefix added to parameter and buffer names to compose the keys in state\_dict. Default: ''.
- **keep\_vars** (*bool*, *optional*) – by default the Tensor s returned in the state dict are detached from autograd. If it's set to True, detaching will not be performed. Default: False.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to(\*args, \*\*kwargs)**

Moves and/or casts the parameters and buffers.

This can be called as

**to**(device=None, dtype=None, non\_blocking=False)

**to**(dtype, non\_blocking=False)

**to**(tensor, non\_blocking=False)

**to**(memory\_format=torch.channels\_last)

Its signature is similar to torch.Tensor.to(), but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to dtype (if given). The integral parameters and buffers will be moved device, if that is given, but with dtypes unchanged.

When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** `self`

**Return type** `Module`

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)
```

(continues on next page)

(continued from previous page)

```

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (type or string) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *device* (int, optional) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**zero\_grad**(*set\_to\_none: bool = False*) → None

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** **set\_to\_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

### 12.1.3 Survival analysis models

#### `synthcity.plugins.core.models.survival_analysis.surv_aft` module

```
class WeibullAFTSurvivalAnalysis(device: Any = device(type='cpu'), **kwargs: Any)
 Bases: synthcity.plugins.core.models.survival_analysis._base.SurvivalAnalysisPlugin

 fit(X: pandas.core.frame.DataFrame, T: pandas.core.series.Series, Y: pandas.core.series.Series) →
 synthcity.plugins.core.models.survival_analysis._base.SurvivalAnalysisPlugin
 Training logic

 static hyperparameter_space(**kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]
 Returns the hyperparameter space for the derived plugin.

 static load(buff: bytes) → Any

 static load_dict(representation: dict) → Any

 static name() → str
 The name of the plugin.

 predict(X: pandas.core.frame.DataFrame, time_horizons: List) → pandas.core.frame.DataFrame
 Predict time-to-event

 classmethod sample_hyperparameters(*args: Any, **kwargs: Any) → Dict[str, Any]
 Sample value from the hyperparameter space for the current plugin.

 save() → bytes

 save_dict() → dict

 save_to_file(path: pathlib.Path) → bytes

 static version() → str
 API version
```

#### `synthcity.plugins.core.models.survival_analysis.surv_coxph` module

```
class CoxPHSurvivalAnalysis(device: Any = device(type='cpu'), alpha: float = 0.05, fit_options: dict =
 {'step_size': 0.1}, **kwargs: Any)
 Bases: synthcity.plugins.core.models.survival_analysis._base.SurvivalAnalysisPlugin

 fit(X: pandas.core.frame.DataFrame, T: pandas.core.series.Series, Y: pandas.core.series.Series) →
 synthcity.plugins.core.models.survival_analysis._base.SurvivalAnalysisPlugin
 Training logic

 static hyperparameter_space(**kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]
 Returns the hyperparameter space for the derived plugin.
```

```
static load(buff: bytes) → Any
static load_dict(representation: dict) → Any
static name() → str
 The name of the plugin.
predict(X: pandas.core.frame.DataFrame, time_horizons: List) → pandas.core.frame.DataFrame
 Predict risk estimation
classmethod sample_hyperparameters(*args: Any, **kwargs: Any) → Dict[str, Any]
 Sample value from the hyperparameter space for the current plugin.
save() → bytes
save_dict() → dict
save_to_file(path: pathlib.Path) → bytes
static version() → str
 API version
```

### **synthcity.plugins.core.models.survival\_analysis.surv\_deephit module**

```
class DeephitSurvivalAnalysis(num_durations: int = 500, batch_size: int = 100, epochs: int = 2000, lr: float
 = 0.01, dim_hidden: int = 300, alpha: float = 0.28, sigma: float = 0.38,
 dropout: float = 0.2, patience: int = 20, batch_norm: bool = False,
 random_state: int = 0, device: Any = device(type='cpu'), **kwargs: Any)
 Bases: synthcity.plugins.core.models.survival_analysis._base.SurvivalAnalysisPlugin
fit(X: pandas.core.frame.DataFrame, T: pandas.core.series.Series, E: pandas.core.series.Series) →
 synthcity.plugins.core.models.survival_analysis._base.SurvivalAnalysisPlugin
 Training logic
static hyperparameter_space(*args: Any, **kwargs: Any) →
 List[synthcity.plugins.core.distribution.Distribution]
 Returns the hyperparameter space for the derived plugin.
static load(buff: bytes) → Any
static load_dict(representation: dict) → Any
static name() → str
 The name of the plugin.
predict(X: pandas.core.frame.DataFrame, time_horizons: List) → pandas.core.frame.DataFrame
 Predict risk
classmethod sample_hyperparameters(*args: Any, **kwargs: Any) → Dict[str, Any]
 Sample value from the hyperparameter space for the current plugin.
save() → bytes
save_dict() → dict
save_to_file(path: pathlib.Path) → bytes
static version() → str
 API version
```

## synthcity.plugins.core.models.survival\_analysis.surv\_xgb module

```

class XGBSurvivalAnalysis(n_estimators: int = 100, colsample_bynode: float = 0.5, max_depth: int = 5,
 subsample: float = 0.5, learning_rate: float = 0.05, min_child_weight: int = 50,
 tree_method: str = 'hist', booster: int = 0, random_state: int = 0, objective: str =
 'aft', strategy: str = 'debiased_bce', bce_n_iter: int = 1000, time_points: int = 100,
 device: Any = device(type='cpu'), **kwargs: Any)

Bases: synthcity.plugins.core.models.survival_analysis._base.SurvivalAnalysisPlugin

booster = ['gbtree', 'gblinear', 'dart']

explain(X: pandas.core.frame.DataFrame) → numpy.ndarray

fit(X: pandas.core.frame.DataFrame, T: pandas.core.series.Series, Y: pandas.core.series.Series) →
 synthcity.plugins.core.models.survival_analysis._base.SurvivalAnalysisPlugin
 Training logic

static hyperparameter_space(**kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]
 Returns the hyperparameter space for the derived plugin.

static load(buff: bytes) → Any

static load_dict(representation: dict) → Any

static name() → str
 The name of the plugin.

predict(X: pandas.core.frame.DataFrame, time_horizons: List) → pandas.core.frame.DataFrame
 Predict risk

classmethod sample_hyperparameters(*args: Any, **kwargs: Any) → Dict[str, Any]
 Sample value from the hyperparameter space for the current plugin.

save() → bytes

save_dict() → dict

save_to_file(path: pathlib.Path) → bytes

static version() → str
 API version

```

### 12.1.4 Time-series survival models

#### synthcity.plugins.core.models.time\_series\_survival.ts\_surv\_coxph module

```

class CoxTimeSeriesSurvival(alpha: float = 0.05, penalizer: float = 0.1, device: Any = device(type='cpu'),
 emb_n_iter: int = 1000, emb_batch_size: int = 100, emb_lr: float = 0.001,
 emb_n_layers_hidden: int = 1, emb_n_units_hidden: int = 40, emb_split: int =
 100, emb_rnn_type: str = 'GRU', emb_output_type: str = 'MLP', emb_alpha:
 float = 0.34, emb_beta: float = 0.27, emb_sigma: float = 0.21, emb_dropout:
 float = 0.06, emb_patience: int = 20, n_iter: Optional[int] = None,
 random_state: int = 0)

Bases: synthcity.plugins.core.models.time_series_survival._base.
TimeSeriesSurvivalPlugin

fit(static: Optional[numpy.ndarray], temporal: numpy.ndarray, observation_times: numpy.ndarray, T:
 numpy.ndarray, E: numpy.ndarray) →
 synthcity.plugins.core.models.time_series_survival._base.TimeSeriesSurvivalPlugin
 Training logic

```

```
static hyperparameter_space(*args: Any, **kwargs: Any) →
 List[synthcity.plugins.core.distribution.Distribution]
 Returns the hyperparameter space for the derived plugin.

static load(buff: bytes) → Any

static load_dict(representation: dict) → Any

static name() → str
 The name of the plugin.

predict(static: Optional[numpy.ndarray], temporal: numpy.ndarray, observation_times: numpy.ndarray,
 time_horizons: List) → numpy.ndarray
 Predict risk

classmethod sample_hyperparameters(*args: Any, **kwargs: Any) → Dict[str, Any]
 Sample value from the hyperparameter space for the current plugin.

save() → bytes

save_dict() → dict

save_to_file(path: pathlib.Path) → bytes

static version() → str
 API version
```

### **synthcity.plugins.core.models.time\_series\_survival.ts\_surv\_dynamic\_deephit module**

```
class DynamicDeepHitLayers(input_dim: int, seq_len: int, output_dim: int, layers_rnn: int, hidden_rnn: int,
 rnn_type: str = 'LSTM', dropout: float = 0.1, risks: int = 1, output_type: str =
 'MLP', device: Any = device(type='cpu'))
```

Bases: torch.nn.modules.module.Module

#### **T\_destination**

alias of TypeVar('T\_destination', bound=Dict[str, Any])

```
add_module(name: str, module: Optional[torch.nn.modules.module.Module]) → None
```

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

#### **Parameters**

- **name** (str) – name of the child module. The child module can be accessed from this module using the given name
- **module** (Module) – child module to be added to the module.

```
apply(fn: Callable[[torch.nn.modules.module.Module], None]) → torch.nn.modules.module.T
```

Applies fn recursively to every submodule (as returned by .children()) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

**Parameters** fn (Module -> None) – function to be applied to each submodule

**Returns** self

**Return type** Module

Example:



```

>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)

```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**buffers**(recurse: bool = True) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (bool) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** torch.Tensor – module buffer

Example:

```

>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)

```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** Module – a child module

**cpu()** → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**double**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to double datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**dump\_patches: bool = False**

**eval**() → torch.nn.modules.module.T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** self

**Return type** Module

**extra\_repr**() → str

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**float**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to float datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**forward**(*x: torch.Tensor*) → torch.Tensor

The forward function that is called when data is passed through DynamicDeepHit.

**forward\_attention**(*x: torch.Tensor, inputmask: torch.Tensor, hidden: torch.Tensor*) → torch.Tensor

**forward\_emb**(*x: torch.Tensor*) → torch.Tensor

The forward function that is called when data is passed through DynamicDeepHit.

**get\_buffer**(*target: str*) → torch.Tensor

Returns the buffer given by **target** if it exists, otherwise throws an error.

See the docstring for **get\_submodule** for a more detailed explanation of this method's functionality as well as how to correctly specify **target**.

**Parameters target** – The fully-qualified string name of the buffer to look for. (See **get\_submodule** for how to specify a fully-qualified string.)

**Returns** The buffer referenced by **target**

**Return type** torch.Tensor

**Raises AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by **target** if it exists, otherwise throws an error.

See the docstring for **get\_submodule** for a more detailed explanation of this method's functionality as well as how to correctly specify **target**.

**Parameters target** – The fully-qualified string name of the Parameter to look for. (See **get\_submodule** for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by **target**

**Return type** torch.nn.Parameter

**Raises AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by **target** if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)
```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by `target`

**Return type** `torch.nn.Module`

**Raises** **`AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**`half()`** → `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**`ipu(device: Optional[Union[int, torch.device]] = None)`** → `torch.nn.modules.module.T`

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (`int`, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**load\_state\_dict**(*state\_dict*: Mapping[str, Any], *strict*: bool = True)

Copies parameters and buffers from *state\_dict* into this module and its descendants. If *strict* is True, then the keys of *state\_dict* must exactly match the keys returned by this module's `state_dict()` function.

#### Parameters

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in *state\_dict* match the keys returned by this module's `state_dict()` function. Default: True

#### Returns

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** NamedTuple with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as None and its corresponding key exists in *state\_dict*, `load_state_dict()` will raise a `RuntimeError`.

---

**modules**() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str*, *torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children()** → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str*, *Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo*: Optional[Set[torch.nn.modules.module.Module]] = None, *prefix*: str = "", *remove\_duplicate*: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str*, *Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str, Parameter*) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → `Iterator[torch.nn.parameter.Parameter]`

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → `torch.utils.hooks.RemovableHandle`

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_buffer**(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → `None`

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm’s `running_mean` is not a parameter, but is part of the module’s state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module’s `state_dict`.

Buffers can be accessed as attributes using given names.

**Parameters**

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as `cuda`, are ignored. If `None`, the buffer is **not** included in the module’s `state_dict`.

- **persistent** (*bool*) – whether the buffer is part of this module’s *state\_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → *torch.utils.hooks.RemovableHandle*

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The *grad\_input* and *grad\_output* are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of *grad\_input* in subsequent computations. *grad\_input* will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in *grad\_input* and *grad\_output* will be *None* for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module’s forward function.



**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Alias for `add_module()`.

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad**(*requires\_grad: bool = True*) → `torch.nn.modules.module.T`

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** `requires_grad` (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** `self`

**Return type** `Module`

**set\_extra\_state**(*state: Any*)

This function is called from `load_state_dict()` to handle any extra state found within the *state\_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state\_dict*.

**Parameters** *state (dict)* – Extra state from the *state\_dict*

**share\_memory**() → torch.nn.modules.module.T

See `torch.Tensor.share_memory_()`

**state\_dict**(\*args, destination=None, prefix="", keep\_vars=False)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently `state_dict()` also accepts positional arguments for *destination*, *prefix* and *keep\_vars* in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument *destination* as it is not designed for end-users.

### Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in *state\_dict*. Default: `''`.
- **keep\_vars** (*bool, optional*) – by default the `Tensor`s returned in the *state dict* are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(\*args, \*\*kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(*device=None, dtype=None, non\_blocking=False*)

**to**(*dtype, non\_blocking=False*)

`to(tensor, non_blocking=False)`

`to(memory_format=torch.channels_last)`

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** `self`

**Return type** `Module`

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```

Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (type or string) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**zero\_grad**(*set\_to\_none: bool = False*) → `None`

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** `set_to_none` (*bool*) – instead of setting to zero, set the grads to `None`. See `torch.optim.Optimizer.zero_grad()` for details.

**class DynamicDeepHitModel**(*split: int = 100, layers\_rnn: int = 2, hidden\_rnn: int = 100, rnn\_type: str = 'LSTM', dropout: float = 0.1, alpha: float = 0.1, beta: float = 0.1, sigma: float = 0.1, patience: int = 20, lr: float = 0.001, batch\_size: int = 100, n\_iter: int = 1000, device: Any = device(type='cpu'), val\_size: float = 0.1, random\_state: int = 0, clipping\_value: int = 1, output\_type: str = 'MLP'*)

Bases: `object`

This implementation considers that the last event happen at the same time for each patient The CIF is therefore simplified

**discretize**(*t: numpy.ndarray, split: int, split\_time: Optional[int] = None*) → `Tuple`

Discretize the survival horizon

**Parameters**

- **t** (*List of Array*) – Time of events
- **split** (*int*) – Number of bins
- **split\_time** (*List, optional*) – List of bins (must be same length than split). Defaults to `None`.

**Returns** Discretized events time

**Return type** `List of Array`

**fit**(*x: numpy.ndarray, t: numpy.ndarray, e: numpy.ndarray*) → `Any`

**longitudinal\_loss**(*longitudinal\_prediction: torch.Tensor, x: torch.Tensor*) → `torch.Tensor`

Penalize error in the longitudinal predictions This function is used to compute the error made by the RNN

NB: In the paper, they seem to use different losses for continuous and categorical But this was not reflected in the code associated (therefore we compute MSE for all)

NB: Original paper mentions possibility of different alphas for each risk But take same for all (for ranking loss)

**negative\_log\_likelihood**(*outcomes: torch.Tensor, cif: torch.Tensor, t: torch.Tensor, e: torch.Tensor*) → `torch.Tensor`

Compute the log likelihood loss This function is used to compute the survival loss

**predict\_emb**(*x: numpy.ndarray*) → `numpy.ndarray`

**predict\_risk**(*x: numpy.ndarray, t: numpy.ndarray, \*\*args: Any*) → `numpy.ndarray`

```
predict_survival(x: numpy.ndarray, t: numpy.ndarray, risk: int = 1, all_step: bool = False, bs: int = 100) → numpy.ndarray

ranking_loss(cif: torch.Tensor, t: torch.Tensor, e: torch.Tensor) → torch.Tensor
 Penalize wrong ordering of probability Equivalent to a C Index This function is used to penalize wrong
 ordering in the survival prediction

total_loss(x: torch.Tensor, t: torch.Tensor, e: torch.Tensor) → torch.Tensor

class DynamicDeephitTimeSeriesSurvival(n_iter: int = 1000, batch_size: int = 100, lr: float = 0.001,
 n_layers_hidden: int = 1, n_units_hidden: int = 40, split: int =
 100, rnn_type: str = 'GRU', alpha: float = 0.34, beta: float =
 0.27, sigma: float = 0.21, random_state: int = 0, dropout: float =
 0.06, device: Any = device(type='cpu'), patience: int = 20,
 output_type: str = 'MLP', **kwargs: Any)

 Bases: synthcity.plugins.core.models.time_series_survival._base.
 TimeSeriesSurvivalPlugin

 fit(static: Optional[numpy.ndarray], temporal: Union[numpy.ndarray, List], observation_times:
 Union[numpy.ndarray, List], T: Union[numpy.ndarray, List], E: Union[numpy.ndarray, List]) →
 synthcity.plugins.core.models.time_series_survival._base.TimeSeriesSurvivalPlugin
 Training logic

 static hyperparameter_space(*args: Any, prefix: str = "", **kwargs: Any) →
 List[synthcity.plugins.core.distribution.Distribution]
 Returns the hyperparameter space for the derived plugin.

 static load(buff: bytes) → Any

 static load_dict(representation: dict) → Any

 static name() → str
 The name of the plugin.

 predict(static: Optional[numpy.ndarray], temporal: Union[numpy.ndarray, List], observation_times:
 Union[numpy.ndarray, List], time_horizons: List, batch_size: Optional[int] = 100) →
 numpy.ndarray
 Predict risk

 predict_emb(static: Optional[numpy.ndarray], temporal: Union[numpy.ndarray, List], observation_times:
 Union[numpy.ndarray, List]) → numpy.ndarray
 Predict embeddings

 classmethod sample_hyperparameters(*args: Any, **kwargs: Any) → Dict[str, Any]
 Sample value from the hyperparameter space for the current plugin.

 save() → bytes

 save_dict() → dict

 save_to_file(path: pathlib.Path) → bytes

 static version() → str
 API version
```

**synthcity.plugins.core.models.time\_series\_survival.ts\_surv\_xgb module**

```
class XGBTimeSeriesSurvival(n_estimators: int = 100, colsample_bynode: float = 0.5, max_depth: int = 5,
 subsample: float = 0.5, learning_rate: float = 0.05, min_child_weight: int = 50,
 tree_method: str = 'hist', booster: int = 0, random_state: int = 0, objective: str
 = 'aft', strategy: str = 'km', bce_n_iter: int = 1000, time_points: int = 100,
 device: Any = device(type='cpu'), emb_n_iter: int = 1000, emb_batch_size: int
 = 100, emb_lr: float = 0.001, emb_n_layers_hidden: int = 1,
 emb_n_units_hidden: int = 40, emb_split: int = 100, emb_rnn_type: str =
 'GRU', emb_output_type: str = 'MLP', emb_alpha: float = 0.34, emb_beta: float
 = 0.27, emb_sigma: float = 0.21, emb_dropout: float = 0.06, emb_patience: int
 = 20, n_iter: Optional[int] = None, **kwargs: Any)
```

Bases: `synthcity.plugins.core.models.time_series_survival._base.`

`TimeSeriesSurvivalPlugin`

```
fit(static: Optional[numpy.ndarray], temporal: numpy.ndarray, observation_times: numpy.ndarray, T:
 numpy.ndarray, E: numpy.ndarray) →
```

`synthcity.plugins.core.models.time_series_survival._base.TimeSeriesSurvivalPlugin`

Training logic

```
static hyperparameter_space(*args: Any, **kwargs: Any) →
 List[synthcity.plugins.core.distribution.Distribution]
```

Returns the hyperparameter space for the derived plugin.

```
static load(buff: bytes) → Any
```

```
static load_dict(representation: dict) → Any
```

```
static name() → str
```

The name of the plugin.

```
predict(static: Optional[numpy.ndarray], temporal: numpy.ndarray, observation_times: numpy.ndarray,
 time_horizons: List) → numpy.ndarray
```

Predict risk

```
classmethod sample_hyperparameters(*args: Any, **kwargs: Any) → Dict[str, Any]
```

Sample value from the hyperparameter space for the current plugin.

```
save() → bytes
```

```
save_dict() → dict
```

```
save_to_file(path: pathlib.Path) → bytes
```

```
static version() → str
```

API version

**12.1.5 Time-to-event models****synthcity.plugins.core.models.time\_to\_event.tte\_date module**

PyTorch implementation for “Adversarial Time-to-Event Modeling” Paper: <https://arxiv.org/pdf/1804.03184.pdf>

```
class DATETimeToEvent(**kwargs: Any)
```

Bases: `synthcity.plugins.core.models.time_to_event._base.TimeToEventPlugin`

```
fit(X: pandas.core.frame.DataFrame, T: pandas.core.series.Series, Y: pandas.core.series.Series) →
```

`synthcity.plugins.core.models.time_to_event._base.TimeToEventPlugin`

Training logic

**static hyperparameter\_space**(\*\*kwargs: Any) → List[synthcity.plugins.core.distribution.Distribution]  
Returns the hyperparameter space for the derived plugin.

**static load**(buff: bytes) → Any

**static load\_dict**(representation: dict) → Any

**static name**() → str  
The name of the plugin.

**predict**(X: pandas.core.frame.DataFrame) → pandas.core.series.Series  
Predict time-to-event

**predict\_any**(X: pandas.core.frame.DataFrame, E: pandas.core.series.Series) → pandas.core.series.Series  
Predict time-to-event

**classmethod sample\_hyperparameters**(\*args: Any, \*\*kwargs: Any) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**save**() → bytes

**save\_dict**() → dict

**save\_to\_file**(path: pathlib.Path) → bytes

**static version**() → str  
API version

**class TimeEventGAN**(n\_features: int, n\_units\_latent: int, model\_search\_n\_iter: Optional[int] = None, generator\_n\_layers\_hidden: int = 2, generator\_n\_units\_hidden: int = 250, generator\_nonlin: str = 'leaky\_relu', generator\_nonlin\_out: Optional[List[Tuple[str, int]]] = None, generator\_n\_iter: int = 1000, generator\_batch\_norm: bool = False, generator\_dropout: float = 0, generator\_loss: Optional[Callable] = None, generator\_lr: float = 0.0002, generator\_weight\_decay: float = 0.001, generator\_residual: bool = True, generator\_opt\_betas: tuple = (0.9, 0.999), discriminator\_n\_layers\_hidden: int = 3, discriminator\_n\_units\_hidden: int = 300, discriminator\_nonlin: str = 'leaky\_relu', discriminator\_n\_iter: int = 1, discriminator\_batch\_norm: bool = False, discriminator\_dropout: float = 0.1, discriminator\_loss: Optional[Callable] = None, discriminator\_lr: float = 0.0002, discriminator\_weight\_decay: float = 0.001, discriminator\_opt\_betas: tuple = (0.9, 0.999), patience: int = 10, batch\_size: int = 100, n\_iter\_print: int = 50, random\_state: int = 0, n\_iter\_min: int = 100, clipping\_value: int = 0, device: Any = device(type='cpu'))

Bases: torch.nn.modules.module.Module

**T\_destination**  
alias of TypeVar('T\_destination', bound=Dict[str, Any])

**add\_module**(name: str, module: Optional[torch.nn.modules.module.Module]) → None  
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

#### Parameters

- **name** (str) – name of the child module. The child module can be accessed from this module using the given name
- **module** (Module) – child module to be added to the module.

**apply**(fn: Callable[[torch.nn.modules.module.Module], None]) → torch.nn.modules.module.T  
Applies fn recursively to every submodule (as returned by .children()) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).



**Parameters** `fn` (Module -> None) – function to be applied to each submodule

**Returns** `self`

**Return type** Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** Module

**buffers** (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module

**cpu()** → torch.nn.modules.module.T  
Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** *Module*

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T  
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** *self*

**Return type** *Module*

**dataloader**(*X: torch.Tensor, T: torch.Tensor, E: torch.Tensor*) →  
Tuple[torch.utils.data.dataloader.DataLoader, torch.utils.data.dataset.TensorDataset]

**double**() → torch.nn.modules.module.T  
Casts all floating point parameters and buffers to double datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** *Module*

**dump\_patches:** **bool = False**

**eval**() → torch.nn.modules.module.T  
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** *self*

**Return type** *Module*

**extra\_repr()** → str

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**fit**(*X: numpy.ndarray, T: numpy.ndarray, E: numpy.ndarray*) → *synthcity.plugins.core.models.time\_to\_event.tte\_date.TimeEventGAN*

**float()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to float datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**forward**(*X: torch.Tensor*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**generate**(*X: numpy.ndarray*) → numpy.ndarray

**get\_buffer**(*target: str*) → torch.Tensor

Returns the buffer given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by *target*

**Return type** torch.Tensor

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by *target*

**Return type** torch.nn.Parameter

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an nn.Parameter

**get\_submodule**(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by *target* if it exists, otherwise throws an error.

For example, let's say you have an nn.Module A that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)
```

(The diagram shows an nn.Module A. A has a nested submodule net\_b, which itself has two submodules net\_c and linear. net\_c then has a submodule conv.)

To check whether or not we have the linear submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the conv submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in *target*. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** *target* – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by *target*

**Return type** torch.nn.Module

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an nn.Module

**half**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to half datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**ipu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *device* (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** *self*

**Return type** Module

**load\_state\_dict**(*state\_dict: Mapping[str, Any]*, *strict: bool = True*)

Copies parameters and buffers from *state\_dict* into this module and its descendants. If *strict* is True, then the keys of *state\_dict* must exactly match the keys returned by this module's *state\_dict()* function.

**Parameters**

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in *state\_dict* match the keys returned by this module's *state\_dict()* function. Default: True

**Returns**

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** NamedTuple with *missing\_keys* and *unexpected\_keys* fields

---

**Note:** If a parameter or buffer is registered as None and its corresponding key exists in *state\_dict*, *load\_state\_dict()* will raise a *RuntimeError*.

---

**modules**() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, 1)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
```

(continues on next page)

(continued from previous page)

```
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str, torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children**() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str, Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo: Optional[Set[torch.nn.modules.module.Module]] = None, prefix: str = "", remove\_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str, Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, 1)
```

(continues on next page)

(continued from previous page)

```
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str, Parameter*) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_buffer**(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm’s `running_mean` is not a parameter, but is part of the module’s state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module’s `state_dict`.

Buffers can be accessed as attributes using given names.

#### Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as `cuda`, are ignored. If `None`, the buffer is **not** included in the module’s `state_dict`.
- **persistent** (*bool*) – whether the buffer is part of this module’s `state_dict`.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle



```
register_full_backward_hook(hook: Callable[[torch.nn.modules.module.Module,
 Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor,
 ...], torch.Tensor]], Union[None, torch.Tensor]]) →
torch.utils.hooks.RemovableHandle
```

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

```
register_load_state_dict_post_hook(hook)
```

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

```
register_module(name: str, module: Optional[torch.nn.modules.module.Module]) → None
```

Alias for `add_module()`.

```
register_parameter(name: str, param: Optional[torch.nn.parameter.Parameter]) → None
```

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

**Parameters**

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If *None*, then operations that run on parameters, such as [cuda](#), are ignored. If *None*, the parameter is **not** included in the module's *state\_dict*.

**requires\_grad\_**(*requires\_grad: bool = True*) → *torch.nn.modules.module.T*

Change if autograd should record operations on parameters in this module.

This method sets the parameters' *requires\_grad* attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See [locally-disable-grad-doc](#) for a comparison between *.requires\_grad\_()* and several similar mechanisms that may be confused with it.

**Parameters** *requires\_grad* (*bool*) – whether autograd should record operations on parameters in this module. Default: *True*.

**Returns** *self*

**Return type** *Module*

**set\_extra\_state**(*state: Any*)

This function is called from [load\\_state\\_dict\(\)](#) to handle any extra state found within the *state\_dict*. Implement this function and a corresponding [get\\_extra\\_state\(\)](#) for your module if you need to store extra state within its *state\_dict*.

**Parameters** *state* (*dict*) – Extra state from the *state\_dict*

**share\_memory**() → *torch.nn.modules.module.T*

See [torch.Tensor.share\\_memory\\_\(\)](#)

**state\_dict**(\**args*, *destination=None*, *prefix=""*, *keep\_vars=False*)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to *None* are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently *state\_dict()* also accepts positional arguments for *destination*, *prefix* and *keep\_vars* in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument *destination* as it is not designed for end-users.

### Parameters

- **destination** (*dict*, *optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an *OrderedDict* will be created and returned. Default: *None*.
- **prefix** (*str*, *optional*) – a prefix added to parameter and buffer names to compose the keys in *state\_dict*. Default: *''*.

- **keep\_vars** (*bool*, *optional*) – by default the Tensor s returned in the state dict are detached from autograd. If it's set to True, detaching will not be performed. Default: False.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(*\*args*, *\*\*kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(*device=None*, *dtype=None*, *non\_blocking=False*)

**to**(*dtype*, *non\_blocking=False*)

**to**(*tensor*, *non\_blocking=False*)

**to**(*memory\_format=torch.channels\_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** self

**Return type** Module

Examples:

```

>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device*: Union[str, torch.device]) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode*: bool = True) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (*type* or *string*) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *device* (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**zero\_grad**(*set\_to\_none: bool = False*) → None

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** *set\_to\_none* (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

## synthcity.plugins.core.models.time\_to\_event.tte\_survival\_function\_regression module

**class** `SurvivalFunctionTimeToEvent`(*time\_points: int = 100, model\_search\_n\_iter: Optional[int] = None, device: Any = device(type='cpu'), \*\*kwargs: Any*)

Bases: `synthcity.plugins.core.models.time_to_event._base.TimeToEventPlugin`

**fit**(*X: pandas.core.frame.DataFrame, T: pandas.core.series.Series, Y: pandas.core.series.Series*) → `synthcity.plugins.core.models.time_to_event._base.TimeToEventPlugin`  
Training logic

**static hyperparameter\_space**(*\*\*kwargs: Any*) → List[`synthcity.plugins.core.distribution.Distribution`]  
Returns the hyperparameter space for the derived plugin.

**static load**(*buff: bytes*) → Any

**static load\_dict**(*representation: dict*) → Any

**static name()** → str  
The name of the plugin.

**predict**(*X: pandas.core.frame.DataFrame*) → pandas.core.series.Series  
Predict time-to-event

**predict\_any**(*X: pandas.core.frame.DataFrame, E: pandas.core.series.Series*) → pandas.core.series.Series  
Predict time-to-event or censoring

**classmethod sample\_hyperparameters**(*\*args: Any, \*\*kwargs: Any*) → Dict[str, Any]  
Sample value from the hyperparameter space for the current plugin.

**save()** → bytes

**save\_dict()** → dict

**save\_to\_file**(*path: pathlib.Path*) → bytes

**static version()** → str  
API version

## 12.1.6 Images

### synthcity.plugins.core.models.convnet module

**class ConditionalDiscriminator**(*model: torch.nn.modules.module.Module, n\_channels: int, height: int, width: int, cond: Optional[torch.Tensor] = None, cond\_embedding\_n\_units\_hidden: int = 100, device: Any = device(type='cpu')*)

Bases: torch.nn.modules.module.Module

#### Parameters

- **model** – nn.Module Core model.
- **n\_channels** – int Number of channels in images
- **height** – int Image height
- **width** – int Image width
- **cond** – torch.Tensor The reference conditional
- **cond\_embedding\_n\_units\_hidden** – int Size of the conditional embedding layer

#### T\_destination

alias of TypeVar('T\_destination', bound=Dict[str, Any])

**add\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

#### Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

**apply**(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** `fn` (Module -> None) – function to be applied to each submodule

**Returns** `self`

**Return type** Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** Module

**buffers** (*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module

**cpu()** → torch.nn.modules.module.T  
Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** *Module*

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T  
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** *self*

**Return type** *Module*

**double()** → torch.nn.modules.module.T  
Casts all floating point parameters and buffers to double datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** *Module*

**dump\_patches:** *bool = False*

**eval()** → torch.nn.modules.module.T  
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** *self*

**Return type** *Module*

**extra\_repr()** → str  
Set the extra representation of the module



To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**float()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to float datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**forward**(*X: torch.Tensor, cond: Optional[torch.Tensor] = None*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**get\_buffer**(*target: str*) → torch.Tensor

Returns the buffer given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by *target*

**Return type** torch.Tensor

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by `target`

**Return type** `torch.nn.Parameter`

**Raises `AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**`get_submodule(target: str)`**  $\rightarrow$  `torch.nn.modules.module.Module`

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)
```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters `target`** – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by `target`

**Return type** `torch.nn.Module`

**Raises `AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**`half()`**  $\rightarrow$  `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**`ipu(device: Optional[Union[int, torch.device]] = None)`**  $\rightarrow$  `torch.nn.modules.module.T`

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**load\_state\_dict**(*state\_dict: Mapping[str, Any]*, *strict: bool = True*)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

**Parameters**

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `True`

**Returns**

- **missing\_keys** is a list of `str` containing the missing keys
- **unexpected\_keys** is a list of `str` containing the unexpected keys

**Return type** `NamedTuple` with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as `None` and its corresponding key exists in `state_dict`, `load_state_dict()` will raise a `RuntimeError`.

---

**modules**() → `Iterator[torch.nn.modules.module.Module]`

Returns an iterator over all modules in the network.

**Yields** `Module` – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, `l` will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix: str = ""*, *recurse: bool = True*) → `Iterator[Tuple[str, torch.Tensor]]`

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

**Parameters**

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if `True`, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str*, *torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children()** → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str*, *Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo*: Optional[Set[torch.nn.modules.module.Module]] = None, *prefix*: str = "", *remove\_duplicate*: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str*, *Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str*, *Parameter*) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_buffer**(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm’s `running_mean` is not a parameter, but is part of the module’s state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to False. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module’s `state_dict`.

Buffers can be accessed as attributes using given names.

### Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name

- **tensor** (*Tensor or None*) – buffer to be registered. If *None*, then operations that run on buffers, such as *cuda*, are ignored. If *None*, the buffer is **not** included in the module's *state\_dict*.
- **persistent** (*bool*) – whether the buffer is part of this module's *state\_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the *forward*. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the *forward*. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → *torch.utils.hooks.RemovableHandle*

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The *grad\_input* and *grad\_output* are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of *grad\_input* in subsequent computations. *grad\_input* will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in *grad\_input* and *grad\_output* will be *None* for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The module argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Alias for `add_module()`.

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad\_**(*requires\_grad: bool = True*) → `torch.nn.modules.module.T`

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See `locally-disable-grad-doc` for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** `requires_grad` (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** self

**Return type** Module

**set\_extra\_state**(*state: Any*)

This function is called from `load_state_dict()` to handle any extra state found within the *state\_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state\_dict*.

**Parameters** **state** (*dict*) – Extra state from the *state\_dict*

**share\_memory**() → torch.nn.modules.module.T

See `torch.Tensor.share_memory_()`

**state\_dict**(\*args, *destination=None, prefix="", keep\_vars=False*)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently `state_dict()` also accepts positional arguments for *destination*, *prefix* and *keep\_vars* in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument *destination* as it is not designed for end-users.

### Parameters

- **destination** (*dict*, *optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str*, *optional*) – a prefix added to parameter and buffer names to compose the keys in *state\_dict*. Default: `''`.
- **keep\_vars** (*bool*, *optional*) – by default the `Tensor`s returned in the *state dict* are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(\*args, \*\*kwargs)

Moves and/or casts the parameters and buffers.

This can be called as



```
to(device=None, dtype=None, non_blocking=False)
```

```
to(dtype, non_blocking=False)
```

```
to(tensor, non_blocking=False)
```

```
to(memory_format=torch.channels_last)
```

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** `self`

**Return type** `Module`

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
```

(continues on next page)

(continued from previous page)

```

 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (type or string) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**zero\_grad**(*set\_to\_none: bool = False*) → `None`

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** `set_to_none` (*bool*) – instead of setting to zero, set the grads to `None`. See `torch.optim.Optimizer.zero_grad()` for details.

**class ConditionalGenerator**(*model: torch.nn.modules.module.Module, n\_channels: int, n\_units\_latent: int, cond: Optional[torch.Tensor] = None, cond\_embedding\_n\_units\_hidden: int = 100, device: Any = device(type='cpu')*)

Bases: `torch.nn.modules.module.Module`

Wrapper for making existing CNN generator conditional. Useful for Conditional GANs

**Parameters**

- **model** – `nn.Module` Core model.
- **n\_channels** – `int` Number of channels in images
- **n\_units\_latent** – `int` Noise size for the input
- **cond** – `torch.Tensor` The reference conditional
- **cond\_embedding\_n\_units\_hidden** – `int` Size of the conditional embedding layer

**T\_destination**

alias of `TypeVar('T_destination', bound=Dict[str, Any])`

**add\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → `None`

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

**Parameters**

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

**apply**(*fn: Callable[[torch.nn.modules.module.Module], None]*) → `torch.nn.modules.module.T`

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as `self`. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** `fn` (*Module -> None*) – function to be applied to each submodule

**Returns** `self`

**Return type** `Module`

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
```

**bfloat16()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**buffers**(recurse: bool = True) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (bool) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** torch.Tensor – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**children()** → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** Module – a child module

**cpu()** → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**double**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to double datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**dump\_patches: bool = False**

**eval**() → torch.nn.modules.module.T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** self

**Return type** Module

**extra\_repr**() → str

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**float**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to float datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**forward**(*noise: torch.Tensor, cond: Optional[torch.Tensor] = None*) → `torch.Tensor`

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**get\_buffer**(*target: str*) → `torch.Tensor`

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

**Parameters** `target` – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by `target`

**Return type** `torch.Tensor`

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state**() → `Any`

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** `object`

**get\_parameter**(*target: str*) → `torch.nn.parameter.Parameter`

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

**Parameters** `target` – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by `target`

**Return type** `torch.nn.Parameter`

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by *target* if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)
```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in *target*. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** *target* – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by *target*

**Return type** torch.nn.Module

**Raises** **AttributeError** – If the *target* string references an invalid path or resolves to something that is not an `nn.Module`

**half**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** Module

**ipu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *device* (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** *self*

**Return type** Module

**load\_state\_dict**(*state\_dict*: Mapping[str, Any], *strict*: bool = True)

Copies parameters and buffers from *state\_dict* into this module and its descendants. If *strict* is True, then the keys of *state\_dict* must exactly match the keys returned by this module's `state_dict()` function.

**Parameters**

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool*, *optional*) – whether to strictly enforce that the keys in *state\_dict* match the keys returned by this module's `state_dict()` function. Default: True

**Returns**

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** NamedTuple with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as `None` and its corresponding key exists in *state\_dict*, `load_state_dict()` will raise a `RuntimeError`.

---

**modules**() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

**Parameters**

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str*, *torch.Tensor*) – Tuple containing the name and buffer

Example:



```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children()** → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str*, *Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo*: Optional[Set[torch.nn.modules.module.Module]] = None, *prefix*: str = "", *remove\_duplicate*: bool = True)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str*, *Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix*: str = "", *recurse*: bool = True) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str, Parameter*) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_buffer**(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm’s `running_mean` is not a parameter, but is part of the module’s state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to False. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module’s `state_dict`.

Buffers can be accessed as attributes using given names.

**Parameters**

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If None, then operations that run on buffers, such as `cuda`, are ignored. If None, the buffer is **not** included in the module’s `state_dict`.

- **persistent** (*bool*) – whether the buffer is part of this module’s *state\_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → *torch.utils.hooks.RemovableHandle*

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won’t be passed to the hooks and only to the *forward*. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling *handle.remove()*

**Return type** *torch.utils.hooks.RemovableHandle*

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → *torch.utils.hooks.RemovableHandle*

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The *grad\_input* and *grad\_output* are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of *grad\_input* in subsequent computations. *grad\_input* will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in *grad\_input* and *grad\_output* will be *None* for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module’s forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Alias for `add_module()`.

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

**requires\_grad\_**(*requires\_grad: bool = True*) → `torch.nn.modules.module.T`

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** `requires_grad` (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

**Returns** `self`

**Return type** `Module`

**set\_extra\_state**(*state: Any*)

This function is called from `load_state_dict()` to handle any extra state found within the *state\_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state\_dict*.

**Parameters** *state (dict)* – Extra state from the *state\_dict*

**share\_memory**() → torch.nn.modules.module.T

See torch.Tensor.share\_memory\_()

**state\_dict**(\*args, destination=None, prefix="", keep\_vars=False)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to None are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently `state_dict()` also accepts positional arguments for *destination*, *prefix* and *keep\_vars* in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument *destination* as it is not designed for end-users.

### Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in *state\_dict*. Default: `''`.
- **keep\_vars** (*bool, optional*) – by default the `Tensor`s returned in the *state dict* are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(\*args, \*\*kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(*device=None, dtype=None, non\_blocking=False*)

**to**(*dtype, non\_blocking=False*)

```
to(tensor, non_blocking=False)
```

```
to(memory_format=torch.channels_last)
```

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** `self`

**Return type** `Module`

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```

Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (type or string) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** `device` (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**zero\_grad**(*set\_to\_none: bool = False*) → `None`

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** `set_to_none` (*bool*) – instead of setting to zero, set the grads to `None`. See `torch.optim.Optimizer.zero_grad()` for details.

**class ConvNet**(*task\_type: str, model: torch.nn.modules.module.Module, lr: float = 0.001, weight\_decay: float = 0.001, opt\_betas: tuple = (0.9, 0.999), n\_iter: int = 1000, batch\_size: int = 500, n\_iter\_print: int = 100, random\_state: int = 0, patience: int = 10, n\_iter\_min: int = 100, clipping\_value: int = 1, early\_stopping: bool = True, device: Any = device(type='cpu')*)

Bases: `torch.nn.modules.module.Module`

Wrapper for convolutional nets for classification and regression.

#### Parameters

- **task\_type** (*str*) – classifier or regression
- **model** (*nn.Module*) – classification or regression model implementation
- **lr** (*float*) – learning rate for optimizer.
- **weight\_decay** (*float*) – l2 (ridge) penalty for the weights.
- **n\_iter** (*int*) – Maximum number of iterations.
- **batch\_size** (*int*) – Batch size
- **n\_iter\_print** (*int*) – Number of iterations after which to print updates and check the validation loss.
- **random\_state** (*int*) – random\_state used
- **patience** (*int*) – Number of iterations to wait before early stopping after decrease in validation loss
- **n\_iter\_min** (*int*) – Minimum number of iterations to go through before starting early stopping
- **clipping\_value** (*int*, *default 1*) – Gradients clipping value
- **early\_stopping** (*bool*) – Enable/disable early stopping

#### T\_destination

alias of `TypeVar('T_destination', bound=Dict[str, Any])`

**add\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → `None`

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

#### Parameters



- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

**apply**(*fn: Callable[[torch.nn.modules.module.Module], None]*) → *torch.nn.modules.module.T*

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** *fn* (*Module* → *None*) – function to be applied to each submodule

**Returns** *self*

**Return type** *Module*

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
```

**bfloat16**() → *torch.nn.modules.module.T*

Casts all floating point parameters and buffers to `bfloat16` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** *self*

**Return type** *Module*

**buffers**(*recurse: bool = True*) → *Iterator[torch.Tensor]*

Returns an iterator over module buffers.

**Parameters** *recurse* (*bool*) – if `True`, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**children()** → Iterator[torch.nn.modules.module.Module]  
Returns an iterator over immediate children modules.

**Yields** *Module* – a child module

**cpu()** → torch.nn.modules.module.T  
Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T  
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**double()** → torch.nn.modules.module.T  
Casts all floating point parameters and buffers to double datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**dump\_patches:** bool = False

**eval()** → torch.nn.modules.module.T  
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** `self`

**Return type** `Module`

**extra\_repr()**  $\rightarrow$  `str`

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**fit**(*X*: `torch.utils.data.dataset.Dataset`)  $\rightarrow$  `synthcity.plugins.core.models.convnet.ConvNet`

**float()**  $\rightarrow$  `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `float` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**forward**(*X*: `torch.Tensor`)  $\rightarrow$  `torch.Tensor`

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**get\_buffer**(*target*: `str`)  $\rightarrow$  `torch.Tensor`

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

**Parameters** `target` – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by `target`

**Return type** `torch.Tensor`

**Raises** **AttributeError** – If the `target` string references an invalid path or resolves to something that is not a buffer

**get\_extra\_state()**  $\rightarrow$  `Any`

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**get\_parameter**(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by *target* if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify *target*.

**Parameters** *target* – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The Parameter referenced by *target*

**Return type** torch.nn.Parameter

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**get\_submodule**(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by *target* if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)
```

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in *target*. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters** *target* – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by *target*

**Return type** torch.nn.Module

**Raises** **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**half**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**ipu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** *self*

**Return type** Module

**load\_state\_dict**(*state\_dict: Mapping[str, Any], strict: bool = True*)

Copies parameters and buffers from *state\_dict* into this module and its descendants. If *strict* is True, then the keys of *state\_dict* must exactly match the keys returned by this module's *state\_dict()* function.

**Parameters**

- **state\_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state\_dict* match the keys returned by this module's *state\_dict()* function. Default: True

**Returns**

- **missing\_keys** is a list of str containing the missing keys
- **unexpected\_keys** is a list of str containing the unexpected keys

**Return type** NamedTuple with *missing\_keys* and *unexpected\_keys* fields

---

**Note:** If a parameter or buffer is registered as None and its corresponding key exists in *state\_dict*, *load\_state\_dict()* will raise a *RuntimeError*.

---

**modules**() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
```

(continues on next page)

(continued from previous page)

```
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str, torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children**() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str, Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo: Optional[Set[torch.nn.modules.module.Module]] = None, prefix: str = "", remove\_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** (*str, Module*) – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, 1)
```

(continues on next page)

(continued from previous page)

```
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** (*str, Parameter*) – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**predict**(*X: torch.Tensor*) → torch.Tensor

**predict\_proba**(*X: torch.Tensor*) → torch.Tensor

**register\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_buffer**(*name: str, tensor: Optional[torch.Tensor], persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

#### Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as `cuda`, are ignored. If `None`, the buffer is **not** included in the module's `state_dict`.
- **persistent** (*bool*) – whether the buffer is part of this module's `state_dict`.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_forward\_pre\_hook**(*hook: Callable[[...], None]*) → `torch.utils.hooks.RemovableHandle`

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`



```
register_full_backward_hook(hook: Callable[[torch.nn.modules.module.Module,
 Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor,
 ...], torch.Tensor]], Union[None, torch.Tensor]]) →
torch.utils.hooks.RemovableHandle
```

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

```
register_load_state_dict_post_hook(hook)
```

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

```
register_module(name: str, module: Optional[torch.nn.modules.module.Module]) → None
```

Alias for `add_module()`.

```
register_parameter(name: str, param: Optional[torch.nn.parameter.Parameter]) → None
```

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

**Parameters**

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If *None*, then operations that run on parameters, such as [cuda](#), are ignored. If *None*, the parameter is **not** included in the module's *state\_dict*.

**requires\_grad\_**(*requires\_grad: bool = True*) → *torch.nn.modules.module.T*

Change if autograd should record operations on parameters in this module.

This method sets the parameters' *requires\_grad* attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See [locally-disable-grad-doc](#) for a comparison between *.requires\_grad\_()* and several similar mechanisms that may be confused with it.

**Parameters** *requires\_grad* (*bool*) – whether autograd should record operations on parameters in this module. Default: *True*.

**Returns** *self*

**Return type** *Module*

**score**(*X: torch.Tensor, y: torch.Tensor*) → *float*

**set\_extra\_state**(*state: Any*)

This function is called from [load\\_state\\_dict\(\)](#) to handle any extra state found within the *state\_dict*. Implement this function and a corresponding [get\\_extra\\_state\(\)](#) for your module if you need to store extra state within its *state\_dict*.

**Parameters** *state* (*dict*) – Extra state from the *state\_dict*

**share\_memory**() → *torch.nn.modules.module.T*

See *torch.Tensor.share\_memory\_()*

**state\_dict**(\*args, *destination=None, prefix="", keep\_vars=False*)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to *None* are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently *state\_dict()* also accepts positional arguments for *destination*, *prefix* and *keep\_vars* in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument *destination* as it is not designed for end-users.

### Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an *OrderedDict* will be created and returned. Default: *None*.

- **prefix**(*str*, *optional*) – a prefix added to parameter and buffer names to compose the keys in `state_dict`. Default: `''`.
- **keep\_vars**(*bool*, *optional*) – by default the `Tensor`s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(\*args, \*\*kwargs)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(*device=None*, *dtype=None*, *non\_blocking=False*)

**to**(*dtype*, *non\_blocking=False*)

**to**(*tensor*, *non\_blocking=False*)

**to**(*memory\_format=torch.channels\_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory\_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** self

**Return type** Module

Examples:

```

>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)

```

**to\_empty**(\**device*: Union[str, torch.device]) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** *device* (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode*: bool = True) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** *mode* (bool) – whether to set training mode (True) or evaluation mode (False).  
Default: True.

**Returns** self

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *dst\_type* (*type* or *string*) – the desired type

**Returns** self

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** *device* (*int*, *optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**zero\_grad**(*set\_to\_none: bool = False*) → None

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

**Parameters** *set\_to\_none* (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

**map\_nonlin**(*nonlin: str*) → <monai.networks.layers.factories.LayerFactory object at 0x7f663d589f40>

**suggest\_image\_classifier\_arch**(*n\_channels: int, height: int, width: int, classes: int, n\_residual\_units: int = 2, nonlin: str = 'prelu', dropout: float = 0.2, last\_nonlin: str = 'softmax', device: Any = device(type='cpu'), strategy: str = 'predefined', lr: float = 0.001, weight\_decay: float = 0.001, opt\_betas: tuple = (0.9, 0.999), n\_iter: int = 1000, batch\_size: int = 500, n\_iter\_print: int = 100, random\_state: int = 0, patience: int = 10, n\_iter\_min: int = 100, clipping\_value: int = 1, early\_stopping: bool = True*) → [synthcity.plugins.core.models.convnet.ConvNet](#)

Helper for selecting compatible architecture for image classifiers.

**Parameters**

- **n\_channels** – int Number of channels in the image
- **height** – int Image height
- **width** – int Image width

- **classes** – int Number of output classes
- **nonlin** – str name of the activation activation layers. Can be relu, elu, prelu or leaky\_relu
- **last\_act** – str output activation
- **dropout** – float = 0.2 Dropout value
- **n\_residual\_units** – int integer stating number of convolutions in residual units, 0 means no residual units
- **device** –  
str PyTorch device. cpu, cuda

#### # Training

**lr: float** learning rate for optimizer.

**weight\_decay: float** l2 (ridge) penalty for the weights.

**n\_iter: int** Maximum number of iterations.

**batch\_size: int** Batch size

**n\_iter\_print: int** Number of iterations after which to print updates and check the validation loss.

**random\_state: int** random\_state used

**patience: int** Number of iterations to wait before early stopping after decrease in validation loss

**n\_iter\_min: int** Minimum number of iterations to go through before starting early stopping

**clipping\_value: int, default 1** Gradients clipping value

**early\_stopping: bool** Enable/disable early stopping

**suggest\_image\_generator\_discriminator\_arch**(*n\_units\_latent: int, n\_channels: int, height: int, width: int, generator\_dropout: float = 0.2, generator\_nonlin: str = 'prelu', generator\_n\_residual\_units: int = 2, discriminator\_dropout: float = 0.2, discriminator\_nonlin: str = 'prelu', discriminator\_n\_residual\_units: int = 2, device: Any = device(type='cpu'), strategy: str = 'predefined', cond: Optional[torch.Tensor] = None, cond\_embedding\_n\_units\_hidden: int = 100*) → Tuple[[synthcity.plugins.core.models.convnet.ConditionalGenerator](#), [synthcity.plugins.core.models.convnet.ConditionalDiscriminator](#)]

Helper for selecting compatible architecture for image generators and discriminators.

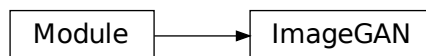
#### Parameters

- **n\_units\_latent** – int, Input size for the generator
- **n\_channels** – int Number of channels in the image
- **height** – int Image height
- **width** – int Image width
- **generator\_dropout** – float = 0.2 Dropout value for the generator

- **generator\_nonlin** – str name of the activation activation layers in the generator. Can be relu, elu, prelu or leaky\_relu
- **generator\_n\_residual\_units** – int integer stating number of convolutions in residual units for the generator, 0 means no residual units
- **discriminator\_dropout** – float = 0.2 Dropout value for the discriminator
- **discriminator\_nonlin** – str name of the activation activation layers in the discriminator. Can be relu, elu, prelu or leaky\_relu
- **discriminator\_n\_residual\_units** – int integer stating number of convolutions in residual units for the discriminator, 0 means no residual units
- **device** – str PyTorch device. cpu, cuda
- **strategy** – str Which suggestion to use. Options:
  - predefined: a few hardcoded architectures for certain image shapes.
  - ...

### synthcity.plugins.core.models.image\_gan module

```
class ImageGAN(image_generator: torch.nn.modules.module.Module, image_discriminator:
 torch.nn.modules.module.Module, n_units_latent: int, n_channels: int, generator_n_iter: int =
 500, generator_lr: float = 0.0002, generator_weight_decay: float = 0.001, generator_opt_betas:
 tuple = (0.9, 0.999), generator_extra_penalties: list = [], generator_extra_penalty_cbks:
 List[Callable] = [], discriminator_n_iter: int = 1, discriminator_lr: float = 0.0002,
 discriminator_weight_decay: float = 0.001, discriminator_opt_betas: tuple = (0.9, 0.999),
 batch_size: int = 100, random_state: int = 0, clipping_value: int = 1, lambda_gradient_penalty:
 float = 10, lambda_identifiability_penalty: float = 0.1, device: Any = device(type='cpu'),
 n_iter_min: int = 100, n_iter_print: int = 1, plot_progress: int = False, patience: int = 20,
 patience_metric: Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None,
 dataloader_sampler: Optional[torch.utils.data.sampler.Sampler] = None, dp_enabled: bool =
 False, dp_delta: Optional[float] = None, dp_epsilon: float = 3, dp_max_grad_norm: float = 2,
 dp_secure_mode: bool = False)
Bases: torch.nn.modules.module.Module
```



Basic GAN implementation.

#### Parameters

- **image\_generator** – nn.Module Generator model
- **image\_discriminator** – nn.Module Discriminator model
- **n\_units\_latent** – int Number of latent units
- **n\_channels** – int Number of channels in the image
- **generator\_n\_iter** – int Maximum number of iterations in the Generator.

- **generator\_lr** – float =  $2e-4$  Generator learning rate, used by the Adam optimizer
- **generator\_weight\_decay** – float =  $1e-3$  Generator weight decay, used by the Adam optimizer
- **generator\_opt\_betas** – tuple = (0.9, 0.999) Generator initial decay rates, used by the Adam Optimizer
- **generator\_extra\_penalties** – list Additional penalties for the generator. Values: “`identifiability_penalty`”
- **generator\_extra\_penalty\_cbks** – List[Callable] Additional loss callabacks for the generator. Used by the TabularGAN for the conditional loss
- **discriminator\_n\_iter** – int Maximum number of iterations in the discriminator.
- **discriminator\_lr** – float Discriminator learning rate, used by the Adam optimizer
- **discriminator\_weight\_decay** – float Discriminator weight decay, used by the Adam optimizer
- **discriminator\_opt\_betas** – tuple Initial weight decays for the Adam optimizer
- **batch\_size** – int Batch size
- **random\_state** – int random\_state used
- **clipping\_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **lambda\_gradient\_penalty** – float = 10 Weight for the gradient penalty
- **lambda\_identifiability\_penalty** – float = 0.1 Weight for the identifiability penalty, if enabled
- **dataloader\_sampler** – Optional[sampler.Sampler] Optional sampler for the dataloader, useful for conditional sampling
- **device** – Any = DEVICE CUDA/CPU
- **stopping** (# *early*) –
- **n\_iter\_print** – int Number of iterations after which to print updates and check the validation loss.
- **n\_iter\_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before early stopping is trigged.
- **patience\_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for early stopping.
- **settings** (# *privacy*) –
- **dp\_enabled** – bool Train the discriminator with Differential Privacy guarantees
- **dp\_delta** – Optional[float] Optional DP delta: the probability of information accidentally being leaked. Usually  $1 / \text{len}(\text{dataset})$
- **dp\_epsilon** – float = 3 DP epsilon: privacy budget, which is a measure of the amount of privacy that is preserved by a given algorithm. Epsilon is a number that represents the maximum amount of information that an adversary can learn about an individual from the output of a differentially private algorithm. The smaller the value of epsilon, the more private the algorithm is. For example, an algorithm with an epsilon of 0.1 preserves more privacy than an algorithm with an epsilon of 1.0.



- **dp\_max\_grad\_norm** – float max grad norm used for gradient clipping
- **dp\_secure\_mode** – bool = False, if True uses noise generation approach robust to floating point arithmetic attacks.

**T\_destination**

alias of `TypeVar('T_destination', bound=Dict[str, Any])`

**add\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

**Parameters**

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

**apply**(*fn: Callable[[torch.nn.modules.module.Module], None]*) → `torch.nn.modules.module.T`

Applies *fn* recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also `nn-init-doc`).

**Parameters** *fn* (*Module* → None) – function to be applied to each submodule

**Returns** self

**Return type** Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>> print(m)
>>> if type(m) == nn.Linear:
>>> m.weight.fill_(1.0)
>>> print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
 [1., 1.]], requires_grad=True)
Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
```

**bfloat16**() → `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to bfloat16 datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**buffers**(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

**Parameters** **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>> print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**children**() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

**Yields** *Module* – a child module

**cpu**() → torch.nn.modules.module.T

Moves all model parameters and buffers to the CPU.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**cuda**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** self

**Return type** Module

**dataloader**(*dataset: torch.utils.data.dataset.Dataset, cond: Optional[torch.Tensor] = None*) → torch.utils.data.dataloader.DataLoader

**double**() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to double datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**dump\_patches: bool = False**

**eval()** → torch.nn.modules.module.T

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between `.eval()` and several similar mechanisms that may be confused with it.

**Returns** self

**Return type** Module

**extra\_repr()** → str

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

**fit**(*X*: synthcity.plugins.core.dataset.FlexibleDataset, *cond*: Optional[torch.Tensor] = None, *fake\_labels\_generator*: Optional[Callable] = None, *true\_labels\_generator*: Optional[Callable] = None) → synthcity.plugins.core.models.image\_gan.ImageGAN

**float()** → torch.nn.modules.module.T

Casts all floating point parameters and buffers to float datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** self

**Return type** Module

**forward**(*count*: int, *cond*: Optional[torch.Tensor] = None) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

---

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

**generate**(*count*: int, *cond*: Optional[torch.Tensor] = None) → torch.Tensor

**get\_buffer**(*target*: str) → torch.Tensor

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

**Parameters** `target` – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The buffer referenced by `target`

**Return type** `torch.Tensor`

**Raises `AttributeError`** – If the target string references an invalid path or resolves to something that is not a buffer

**`get_extra_state()`** → Any

Returns any extra state to include in the module's `state_dict`. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's `state_dict()`.

Note that extra state should be pickleable to ensure working serialization of the `state_dict`. We only provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

**Returns** Any extra state to store in the module's `state_dict`

**Return type** object

**`get_parameter(target: str)`** → `torch.nn.parameter.Parameter`

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

**Parameters `target`** – The fully-qualified string name of the `Parameter` to look for. (See `get_submodule` for how to specify a fully-qualified string.)

**Returns** The `Parameter` referenced by `target`

**Return type** `torch.nn.Parameter`

**Raises `AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

**`get_submodule(target: str)`** → `torch.nn.modules.module.Module`

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` A that looks like this:

```
A(
 (net_b): Module(
 (net_c): Module(
 (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
)
 (linear): Linear(in_features=100, out_features=200, bias=True)
)
)
```

(The diagram shows an `nn.Module` A. A has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is  $O(N)$  in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

**Parameters `target`** – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

**Returns** The submodule referenced by `target`

**Return type** `torch.nn.Module`

**Raises `AttributeError`** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

**`half()`**  $\rightarrow$  `torch.nn.modules.module.T`

Casts all floating point parameters and buffers to `half` datatype.

---

**Note:** This method modifies the module in-place.

---

**Returns** `self`

**Return type** `Module`

**`ipu(device: Optional[Union[int, torch.device]] = None)`**  $\rightarrow$  `torch.nn.modules.module.T`

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters `device` (`int`, `optional`)** – if specified, all parameters will be copied to that device

**Returns** `self`

**Return type** `Module`

**`load_state_dict(state_dict: Mapping[str, Any], strict: bool = True)`**

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

**Parameters**

- **`state_dict` (`dict`)** – a dict containing parameters and persistent buffers.
- **`strict` (`bool`, `optional`)** – whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `True`

**Returns**

- **`missing_keys`** is a list of `str` containing the missing keys
- **`unexpected_keys`** is a list of `str` containing the unexpected keys

**Return type** `NamedTuple` with `missing_keys` and `unexpected_keys` fields

---

**Note:** If a parameter or buffer is registered as `None` and its corresponding key exists in `state_dict`, `load_state_dict()` will raise a `RuntimeError`.

---

**`modules()`**  $\rightarrow$  `Iterator[torch.nn.modules.module.Module]`

Returns an iterator over all modules in the network.

**Yields** *Module* – a module in the network

**Note:** Duplicate modules are returned only once. In the following example, 1 will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(1, l)
>>> for idx, m in enumerate(net.modules()):
... print(idx, '->', m)

0 -> Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

**named\_buffers**(*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

**Yields** (*str, torch.Tensor*) – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>> if name in ['running_var']:
>>> print(buf.size())
```

**named\_children**() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**Yields** (*str, Module*) – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

**named\_modules**(*memo: Optional[Set[torch.nn.modules.module.Module]] = None, prefix: str = "", remove\_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

#### Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module

- **remove\_duplicate** – whether to remove the duplicated module instances in the result or not

**Yields** *(str, Module)* – Tuple of name and module

---

**Note:** Duplicate modules are returned only once. In the following example, `l` will be returned only once.

---

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
... print(idx, '->', m)

0 -> ('', Sequential(
 (0): Linear(in_features=2, out_features=2, bias=True)
 (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

**named\_parameters**(*prefix: str = "", recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

#### Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

**parameters**(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

**Parameters** **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

**Yields** *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>> print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

**register\_backward\_hook**(hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_buffer**(name: str, tensor: Optional[torch.Tensor], persistent: bool = True) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to False. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

#### Parameters

- **name** (str) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (Tensor or None) – buffer to be registered. If None, then operations that run on buffers, such as `cuda`, are ignored. If None, the buffer is **not** included in the module's `state_dict`.
- **persistent** (bool) – whether the buffer is part of this module's `state_dict`.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

**register\_forward\_hook**(hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after `forward()` is called.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** torch.utils.hooks.RemovableHandle

**register\_forward\_pre\_hook**(hook: Callable[[...], None]) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before `forward()` is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```



The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned (unless that value is already a tuple).

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_full\_backward\_hook**(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → `torch.utils.hooks.RemovableHandle`

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

**Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_load\_state\_dict\_post\_hook**(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

**It should have the following signature::** `hook(module, incompatible_keys) -> None`

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a list of `str` containing the missing keys and `unexpected_keys` is a list of `str` containing the unexpected keys.

The given `incompatible_keys` can be modified inplace if needed.

Note that the checks performed when calling `load_state_dict()` with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

**Returns** a handle that can be used to remove the added hook by calling `handle.remove()`

**Return type** `torch.utils.hooks.RemovableHandle`

**register\_module**(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None  
Alias for [add\\_module\(\)](#).

**register\_parameter**(*name: str, param: Optional[torch.nn.parameter.Parameter]*) → None  
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

#### Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If *None*, then operations that run on parameters, such as [cuda](#), are ignored. If *None*, the parameter is **not** included in the module's [state\\_dict](#).

**requires\_grad\_**(*requires\_grad: bool = True*) → torch.nn.modules.module.T  
Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between `.requires_grad_()` and several similar mechanisms that may be confused with it.

**Parameters** **requires\_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: *True*.

**Returns** *self*

**Return type** Module

**set\_extra\_state**(*state: Any*)  
This function is called from [load\\_state\\_dict\(\)](#) to handle any extra state found within the *state\_dict*. Implement this function and a corresponding [get\\_extra\\_state\(\)](#) for your module if you need to store extra state within its *state\_dict*.

**Parameters** **state** (*dict*) – Extra state from the *state\_dict*

**share\_memory**() → torch.nn.modules.module.T  
See `torch.Tensor.share_memory_()`

**state\_dict**(\*args, *destination=None, prefix="", keep\_vars=False*)  
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to *None* are not included.

---

**Note:** The returned object is a shallow copy. It contains references to the module's parameters and buffers.

---

**Warning:** Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

**Warning:** Please avoid the use of argument `destination` as it is not designed for end-users.

### Parameters

- **destination** (*dict*, *optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str*, *optional*) – a prefix added to parameter and buffer names to compose the keys in `state_dict`. Default: `''`.
- **keep\_vars** (*bool*, *optional*) – by default the `Tensor`s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

**Returns** a dictionary containing a whole state of the module

**Return type** `dict`

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

**to**(*\*args*, *\*\*kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

**to**(*device=None*, *dtype=None*, *non\_blocking=False*)

**to**(*dtype*, *non\_blocking=False*)

**to**(*tensor*, *non\_blocking=False*)

**to**(*memory\_format=torch.channels\_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

---

**Note:** This method modifies the module in-place.

---

### Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

- **memory\_format** (torch.memory\_format) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

**Returns** self

**Return type** Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1913, -0.3420],
 [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[0.1914, -0.3420],
 [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[0.3741+0.j, 0.2382+0.j],
 [0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j],
 [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

**to\_empty**(\*, device: Union[str, torch.device]) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

**Parameters** **device** (torch.device) – The desired device of the parameters and buffers in this module.

**Returns** self

**Return type** Module

**train**(*mode: bool = True*) → torch.nn.modules.module.T

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

**Parameters** **mode** (*bool*) – whether to set training mode (True) or evaluation mode (False).

Default: True.

**Returns** *self*

**Return type** Module

**training:** bool

**type**(*dst\_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T

Casts all parameters and buffers to *dst\_type*.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **dst\_type** (*type or string*) – the desired type

**Returns** *self*

**Return type** Module

**xpu**(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

---

**Note:** This method modifies the module in-place.

---

**Parameters** **device** (*int, optional*) – if specified, all parameters will be copied to that device

**Returns** *self*

**Return type** Module

**zero\_grad**(*set\_to\_none: bool = False*) → None

Sets gradients of all model parameters to zero. See similar function under torch.optim.Optimizer for more context.

**Parameters** **set\_to\_none** (*bool*) – instead of setting to zero, set the grads to None. See torch.optim.Optimizer.zero\_grad() for details.

**display\_imgs**(*imgs: List[numpy.ndarray], title: Optional[str] = None*) → None

**weights\_init**(*m: torch.nn.modules.module.Module*) → None



## PYTHON MODULE INDEX

### S

`synthcity.benchmark`, 189  
`synthcity.benchmark.utils`, 190  
`synthcity.metrics.eval_attacks`, 187  
`synthcity.metrics.eval_detection`, 178  
`synthcity.metrics.eval_performance`, 174  
`synthcity.metrics.eval_privacy`, 181  
`synthcity.metrics.eval_sanity`, 165  
`synthcity.metrics.eval_statistical`, 168  
`synthcity.metrics.weighted_metrics`, 189  
`synthcity.plugins.core.constraints`, 197  
`synthcity.plugins.core.dataloader`, 29  
`synthcity.plugins.core.dataset`, 40  
`synthcity.plugins.core.distribution`, 199  
`synthcity.plugins.core.models.convnet`, 474  
`synthcity.plugins.core.models.flows`, 261  
`synthcity.plugins.core.models.gan`, 246  
`synthcity.plugins.core.models.image_gan`, 515  
`synthcity.plugins.core.models.mlp`, 219  
`synthcity.plugins.core.models.survival_analysis`, 441  
`synthcity.plugins.core.models.survival_analysis.surv_arf`, 441  
`synthcity.plugins.core.models.survival_analysis.surv_coxph`, 441  
`synthcity.plugins.core.models.survival_analysis.surv_deephit`, 442  
`synthcity.plugins.core.models.survival_analysis.surv_ign`, 443  
`synthcity.plugins.core.models.tabular_encoder`, 215  
`synthcity.plugins.core.models.tabular_flows`, 358  
`synthcity.plugins.core.models.tabular_gan`, 328  
`synthcity.plugins.core.models.tabular_vae`, 343  
`synthcity.plugins.core.models.time_series_survival`, 443  
`synthcity.plugins.core.models.time_series_survival.ts_sur_coxph`, 444  
`synthcity.plugins.core.models.time_series_survival.ts_sur_dynamic_deephit`, 444  
`synthcity.plugins.core.models.time_series_survival.ts_sur_ign`, 459  
`synthcity.plugins.core.models.time_to_event.tte_date`, 459  
`synthcity.plugins.core.models.time_to_event.tte_survival_flow`, 473  
`synthcity.plugins.core.models.transformer`, 315  
`synthcity.plugins.core.models.ts_gan`, 372  
`synthcity.plugins.core.models.ts_vae`, 387  
`synthcity.plugins.core.models.vae`, 275  
`synthcity.plugins.core.plugin`, 193  
`synthcity.plugins.core.schema`, 213  
`synthcity.plugins.core.serializable`, 215  
`synthcity.plugins.domain_adaptation.plugin_radialgan`, 90  
`synthcity.plugins.generic.plugin_arf`, 24  
`synthcity.plugins.generic.plugin_bayesian_network`, 43  
`synthcity.plugins.generic.plugin_ctgan`, 47  
`synthcity.plugins.generic.plugin_goggle`, 21  
`synthcity.plugins.generic.plugin_nflow`, 52  
`synthcity.plugins.generic.plugin_rtvae`, 57  
`synthcity.plugins.generic.plugin_tvae`, 61  
`synthcity.plugins.images.plugin_image_adsgan`, 158  
`synthcity.plugins.images.plugin_image_cgan`, 154  
`synthcity.plugins.privacy.plugin_adsgan`, 65  
`synthcity.plugins.privacy.plugin_decaf`, 85  
`synthcity.plugins.privacy.plugin_dpgan`, 81  
`synthcity.plugins.privacy.plugin_pategan`, 70  
`synthcity.plugins.privacy.plugin_privbayes`, 76  
`synthcity.plugins.survival_analysis.plugin_survae`, 131  
`synthcity.plugins.survival_analysis.plugin_survival_ctgan`, 127  
`synthcity.plugins.survival_analysis.plugin_survival_gan`, 123  
`synthcity.plugins.survival_analysis.plugin_survival_nflow`, 135  
`synthcity.plugins.time_series.plugin_fflows`, 144  
`synthcity.plugins.time_series.plugin_timegan`, 144

[139](#)  
`synthcity.plugins.time_series.plugin_timevae,`  
[149](#)



## A

- `activation_layout()` (*BinEncoder* method), 215
- `activation_layout()` (*TabularEncoder* method), 217
- `activation_layout()` (*TimeSeriesTabularEncoder* method), 218
- `activation_layout_temporal()` (*TimeSeriesTabularEncoder* method), 218
- `adapt_dtypes()` (*Schema* method), 213
- `add()` (*PluginLoader* method), 196
- `add_module()` (*ConditionalDiscriminator* method), 474
- `add_module()` (*ConditionalGenerator* method), 487
- `add_module()` (*ConvNet* method), 500
- `add_module()` (*Decoder* method), 275
- `add_module()` (*DynamicDeepHitLayers* method), 444
- `add_module()` (*Encoder* method), 288
- `add_module()` (*GAN* method), 248
- `add_module()` (*ImageGAN* method), 517
- `add_module()` (*LatentODE* method), 387
- `add_module()` (*LinearLayer* method), 219
- `add_module()` (*MLP* method), 233
- `add_module()` (*NormalizingFlows* method), 262
- `add_module()` (*RadialGAN* method), 92
- `add_module()` (*TabularFlows* method), 359
- `add_module()` (*TabularGAN* method), 330
- `add_module()` (*TabularRadialGAN* method), 110
- `add_module()` (*TabularVAE* method), 345
- `add_module()` (*TimeEventGAN* method), 460
- `add_module()` (*TimeSeriesDecoder* method), 400
- `add_module()` (*TimeSeriesEncoder* method), 413
- `add_module()` (*TimeSeriesGAN* method), 374
- `add_module()` (*TimeSeriesVAE* method), 428
- `add_module()` (*TransformerModel* method), 315
- `add_module()` (*VAE* method), 302
- `adjust_inference_sampling()` (*TabularGAN* method), 330
- `AdsGANPlugin` (class in *synthetic.plugins.privacy.plugin\_adsgan*), 65
- `AdsGANPlugin.Config` (class in *synthetic.plugins.privacy.plugin\_adsgan*), 67
- `AlphaPrecision` (class in *synthetic.metrics.eval\_statistical*), 168
- `apply()` (*ConditionalDiscriminator* method), 474
- `apply()` (*ConditionalGenerator* method), 487
- `apply()` (*ConvNet* method), 501
- `apply()` (*Decoder* method), 275
- `apply()` (*DynamicDeepHitLayers* method), 444
- `apply()` (*Encoder* method), 288
- `apply()` (*GAN* method), 248
- `apply()` (*ImageGAN* method), 517
- `apply()` (*LatentODE* method), 387
- `apply()` (*LinearLayer* method), 219
- `apply()` (*MLP* method), 233
- `apply()` (*NormalizingFlows* method), 262
- `apply()` (*RadialGAN* method), 92
- `apply()` (*TabularFlows* method), 359
- `apply()` (*TabularGAN* method), 330
- `apply()` (*TabularRadialGAN* method), 110
- `apply()` (*TabularVAE* method), 345
- `apply()` (*TimeEventGAN* method), 460
- `apply()` (*TimeSeriesDecoder* method), 400
- `apply()` (*TimeSeriesEncoder* method), 413
- `apply()` (*TimeSeriesGAN* method), 374
- `apply()` (*TimeSeriesVAE* method), 428
- `apply()` (*TransformerModel* method), 315
- `apply()` (*VAE* method), 302
- `arbitrary_types_allowed` (*AdsGANPlugin.Config* attribute), 67
- `arbitrary_types_allowed` (*ARFPlugin.Config* attribute), 25
- `arbitrary_types_allowed` (*BayesianNetworkPlugin.Config* attribute), 44
- `arbitrary_types_allowed` (*CategoricalDistribution.Config* attribute), 200
- `arbitrary_types_allowed` (*CTGANPlugin.Config* attribute), 49
- `arbitrary_types_allowed` (*DatetimeDistribution.Config* attribute), 201
- `arbitrary_types_allowed` (*DECAFPlugin.Config* attribute), 87
- `arbitrary_types_allowed` (*Distribution.Config* attribute), 204
- `arbitrary_types_allowed` (*DPGANPlugin.Config* attribute), 83
- `arbitrary_types_allowed` (*FloatDistribution.Config* attribute), 201

attribute), 205  
 arbitrary\_types\_allowed (FourierFlowsPlugin.Config attribute), 146  
 arbitrary\_types\_allowed (GOGGLEPlugin.Config attribute), 22  
 arbitrary\_types\_allowed (ImageAdsGANPlugin.Config attribute), 160  
 arbitrary\_types\_allowed (ImageCGANPlugin.Config attribute), 155  
 arbitrary\_types\_allowed (IntegerDistribution.Config attribute), 209  
 arbitrary\_types\_allowed (IntLogDistribution.Config attribute), 207  
 arbitrary\_types\_allowed (LogDistribution.Config attribute), 211  
 arbitrary\_types\_allowed (NormalizingFlowsPlugin.Config attribute), 54  
 arbitrary\_types\_allowed (PATEGANPlugin.Config attribute), 72  
 arbitrary\_types\_allowed (Plugin.Config attribute), 194  
 arbitrary\_types\_allowed (PrivBayesPlugin.Config attribute), 77  
 arbitrary\_types\_allowed (RadialGANPlugin.Config attribute), 106  
 arbitrary\_types\_allowed (RTVAEPlugin.Config attribute), 58  
 arbitrary\_types\_allowed (SurVAEPlugin.Config attribute), 132  
 arbitrary\_types\_allowed (SurvivalCTGANPlugin.Config attribute), 128  
 arbitrary\_types\_allowed (SurvivalGANPlugin.Config attribute), 124  
 arbitrary\_types\_allowed (SurvivalNFlowPlugin.Config attribute), 136  
 arbitrary\_types\_allowed (TimeGANPlugin.Config attribute), 141  
 arbitrary\_types\_allowed (TimeVAEPlugin.Config attribute), 151  
 arbitrary\_types\_allowed (TVAEPlugin.Config attribute), 62  
 ARFPlugin (class in synthcity.plugins.generic.plugin\_arf), 24  
 ARFPlugin.Config (class in synthcity.plugins.generic.plugin\_arf), 25  
 as\_constraint() (CategoricalDistribution method), 200  
 as\_constraint() (DatetimeDistribution method), 201  
 as\_constraint() (Distribution method), 204  
 as\_constraint() (FloatDistribution method), 205  
 as\_constraint() (IntegerDistribution method), 209  
 as\_constraint() (IntLogDistribution method), 207  
 as\_constraint() (LogDistribution method), 211  
 as\_constraints() (Schema method), 213

AttackEvaluator (class in synthcity.metrics.eval\_attacks), 187  
 augment\_data() (in module synthcity.benchmark.utils), 190  
 AugmentationPerformanceEvaluatorLinear (class in synthcity.metrics.eval\_performance), 174  
 AugmentationPerformanceEvaluatorMLP (class in synthcity.metrics.eval\_performance), 175  
 AugmentationPerformanceEvaluatorXGB (class in synthcity.metrics.eval\_performance), 175

## B

BasicMetricEvaluator (class in synthcity.metrics.eval\_sanity), 165  
 BayesianNetworkPlugin (class in synthcity.plugins.generic.plugin\_bayesian\_network), 43  
 BayesianNetworkPlugin.Config (class in synthcity.plugins.generic.plugin\_bayesian\_network), 44  
 Benchmarks (class in synthcity.benchmark), 189  
 bfloat16() (ConditionalDiscriminator method), 475  
 bfloat16() (ConditionalGenerator method), 488  
 bfloat16() (ConvNet method), 501  
 bfloat16() (Decoder method), 276  
 bfloat16() (DynamicDeepHitLayers method), 445  
 bfloat16() (Encoder method), 289  
 bfloat16() (GAN method), 249  
 bfloat16() (ImageGAN method), 517  
 bfloat16() (LatentODE method), 388  
 bfloat16() (LinearLayer method), 220  
 bfloat16() (MLP method), 234  
 bfloat16() (NormalizingFlows method), 263  
 bfloat16() (RadialGAN method), 92  
 bfloat16() (TabularFlows method), 360  
 bfloat16() (TabularGAN method), 331  
 bfloat16() (TabularRadialGAN method), 110  
 bfloat16() (TabularVAE method), 345  
 bfloat16() (TimeEventGAN method), 461  
 bfloat16() (TimeSeriesDecoder method), 401  
 bfloat16() (TimeSeriesEncoder method), 414  
 bfloat16() (TimeSeriesGAN method), 375  
 bfloat16() (TimeSeriesVAE method), 429  
 bfloat16() (TransformerModel method), 316  
 bfloat16() (VAE method), 303  
 BinEncoder (class in synthcity.plugins.core.models.tabular\_encoder), 215  
 booster (XGBSurvivalAnalysis attribute), 443  
 buffers() (ConditionalDiscriminator method), 475  
 buffers() (ConditionalGenerator method), 488  
 buffers() (ConvNet method), 501  
 buffers() (Decoder method), 276  
 buffers() (DynamicDeepHitLayers method), 445

[buffers\(\)](#) (*Encoder method*), 289  
[buffers\(\)](#) (*GAN method*), 249  
[buffers\(\)](#) (*ImageGAN method*), 518  
[buffers\(\)](#) (*LatentODE method*), 388  
[buffers\(\)](#) (*LinearLayer method*), 220  
[buffers\(\)](#) (*MLP method*), 234  
[buffers\(\)](#) (*NormalizingFlows method*), 263  
[buffers\(\)](#) (*RadialGAN method*), 93  
[buffers\(\)](#) (*TabularFlows method*), 360  
[buffers\(\)](#) (*TabularGAN method*), 331  
[buffers\(\)](#) (*TabularRadialGAN method*), 111  
[buffers\(\)](#) (*TabularVAE method*), 346  
[buffers\(\)](#) (*TimeEventGAN method*), 461  
[buffers\(\)](#) (*TimeSeriesDecoder method*), 401  
[buffers\(\)](#) (*TimeSeriesEncoder method*), 414  
[buffers\(\)](#) (*TimeSeriesGAN method*), 375  
[buffers\(\)](#) (*TimeSeriesVAE method*), 429  
[buffers\(\)](#) (*TransformerModel method*), 316  
[buffers\(\)](#) (*VAE method*), 303

## C

[calculate\\_fair\\_aug\\_sample\\_size\(\)](#) (*in module synthcity.benchmark.utils*), 191  
[cat\\_encoder\\_params](#) (*BinEncoder attribute*), 215  
[cat\\_encoder\\_params](#) (*TabularEncoder attribute*), 217  
[categorical\\_encoder](#) (*BinEncoder attribute*), 215  
[categorical\\_encoder](#) (*TabularEncoder attribute*), 217  
[CategoricalDistribution](#) (*class in synthcity.plugins.core.distribution*), 199  
[CategoricalDistribution.Config](#) (*class in synthcity.plugins.core.distribution*), 199  
[children\(\)](#) (*ConditionalDiscriminator method*), 475  
[children\(\)](#) (*ConditionalGenerator method*), 488  
[children\(\)](#) (*ConvNet method*), 502  
[children\(\)](#) (*Decoder method*), 276  
[children\(\)](#) (*DynamicDeepHitLayers method*), 445  
[children\(\)](#) (*Encoder method*), 289  
[children\(\)](#) (*GAN method*), 249  
[children\(\)](#) (*ImageGAN method*), 518  
[children\(\)](#) (*LatentODE method*), 388  
[children\(\)](#) (*LinearLayer method*), 220  
[children\(\)](#) (*MLP method*), 234  
[children\(\)](#) (*NormalizingFlows method*), 263  
[children\(\)](#) (*RadialGAN method*), 93  
[children\(\)](#) (*TabularFlows method*), 360  
[children\(\)](#) (*TabularGAN method*), 331  
[children\(\)](#) (*TabularRadialGAN method*), 111  
[children\(\)](#) (*TabularVAE method*), 346  
[children\(\)](#) (*TimeEventGAN method*), 461  
[children\(\)](#) (*TimeSeriesDecoder method*), 401  
[children\(\)](#) (*TimeSeriesEncoder method*), 414  
[children\(\)](#) (*TimeSeriesGAN method*), 375  
[children\(\)](#) (*TimeSeriesVAE method*), 429  
[children\(\)](#) (*TransformerModel method*), 316  
[children\(\)](#) (*VAE method*), 303  
[ChiSquaredTest](#) (*class in synthcity.metrics.eval\_statistical*), 169  
[choices](#) (*CategoricalDistribution attribute*), 200  
[CloseValuesProbability](#) (*class in synthcity.metrics.eval\_sanitiy*), 165  
[columns](#) (*DataLoader property*), 30  
[columns](#) (*GenericDataLoader property*), 31  
[columns](#) (*ImageDataLoader property*), 33  
[columns](#) (*SurvivalAnalysisDataLoader property*), 34  
[columns](#) (*TimeSeriesDataLoader property*), 36  
[columns](#) (*TimeSeriesSurvivalDataLoader property*), 38  
[CommonRowsProportion](#) (*class in synthcity.metrics.eval\_sanitiy*), 166  
[compress\(\)](#) (*DataLoader method*), 30  
[compress\(\)](#) (*GenericDataLoader method*), 31  
[compress\(\)](#) (*ImageDataLoader method*), 33  
[compress\(\)](#) (*SurvivalAnalysisDataLoader method*), 34  
[compress\(\)](#) (*TimeSeriesDataLoader method*), 36  
[compress\(\)](#) (*TimeSeriesSurvivalDataLoader method*), 38  
[compression\\_protected\\_features\(\)](#) (*DataLoader method*), 30  
[compression\\_protected\\_features\(\)](#) (*GenericDataLoader method*), 31  
[compression\\_protected\\_features\(\)](#) (*ImageDataLoader method*), 33  
[compression\\_protected\\_features\(\)](#) (*SurvivalAnalysisDataLoader method*), 34  
[compression\\_protected\\_features\(\)](#) (*TimeSeriesDataLoader method*), 36  
[compression\\_protected\\_features\(\)](#) (*TimeSeriesSurvivalDataLoader method*), 38  
[ConditionalDataset](#) (*class in synthcity.plugins.core.dataset*), 40  
[ConditionalDiscriminator](#) (*class in synthcity.plugins.core.models.convnet*), 474  
[ConditionalGenerator](#) (*class in synthcity.plugins.core.models.convnet*), 487  
[Config](#) (*Constraints attribute*), 197  
[Config](#) (*FeatureInfo attribute*), 216  
[Config](#) (*Schema attribute*), 213  
[constraint\\_to\\_distribution\(\)](#) (*in module synthcity.plugins.core.distribution*), 212  
[Constraints](#) (*class in synthcity.plugins.core.constraints*), 197  
[construct\(\)](#) (*CategoricalDistribution class method*), 200  
[construct\(\)](#) (*Constraints class method*), 197  
[construct\(\)](#) (*DatetimeDistribution class method*), 202  
[construct\(\)](#) (*Distribution class method*), 204  
[construct\(\)](#) (*FeatureInfo class method*), 216  
[construct\(\)](#) (*FloatDistribution class method*), 205  
[construct\(\)](#) (*IntegerDistribution class method*), 209

- construct() (*IntLogDistribution* class method), 207  
 construct() (*LogDistribution* class method), 211  
 construct() (*Schema* class method), 213  
 cont\_encoder\_params (*BinEncoder* attribute), 215  
 cont\_encoder\_params (*TabularEncoder* attribute), 217  
 continuous\_encoder (*BinEncoder* attribute), 215  
 continuous\_encoder (*TabularEncoder* attribute), 217  
 ConvNet (class in *synthcity.plugins.core.models.convnet*), 500  
 copy() (*CategoricalDistribution* method), 200  
 copy() (*Constraints* method), 197  
 copy() (*DatetimeDistribution* method), 202  
 copy() (*Distribution* method), 204  
 copy() (*FeatureInfo* method), 216  
 copy() (*FloatDistribution* method), 206  
 copy() (*IntegerDistribution* method), 209  
 copy() (*IntLogDistribution* method), 207  
 copy() (*LogDistribution* method), 211  
 copy() (*Schema* method), 213  
 count() (*network\_edge* method), 80  
 CoxPHSurvivalAnalysis (class in *synthcity.plugins.core.models.survival\_analysis.surv\_coxph*), 441  
 CoxTimeSeriesSurvival (class in *synthcity.plugins.core.models.time\_series\_survival.ts\_surv\_coxph*), 443  
 cpu() (*ConditionalDiscriminator* method), 476  
 cpu() (*ConditionalGenerator* method), 488  
 cpu() (*ConvNet* method), 502  
 cpu() (*Decoder* method), 276  
 cpu() (*DynamicDeepHitLayers* method), 445  
 cpu() (*Encoder* method), 289  
 cpu() (*GAN* method), 249  
 cpu() (*ImageGAN* method), 518  
 cpu() (*LatentODE* method), 388  
 cpu() (*LinearLayer* method), 220  
 cpu() (*MLP* method), 234  
 cpu() (*NormalizingFlows* method), 263  
 cpu() (*RadialGAN* method), 93  
 cpu() (*TabularFlows* method), 360  
 cpu() (*TabularGAN* method), 332  
 cpu() (*TabularRadialGAN* method), 111  
 cpu() (*TabularVAE* method), 346  
 cpu() (*TimeEventGAN* method), 462  
 cpu() (*TimeSeriesDecoder* method), 401  
 cpu() (*TimeSeriesEncoder* method), 414  
 cpu() (*TimeSeriesGAN* method), 375  
 cpu() (*TimeSeriesVAE* method), 429  
 cpu() (*TransformerModel* method), 316  
 cpu() (*VAE* method), 303  
 create\_alternating\_binary\_mask() (in module *synthcity.plugins.core.models.flows*), 275  
 create\_from\_info() (in module *synthcity.plugins.core.data\_loader*), 40  
 CTGANPlugin (class in *synthcity.plugins.generic.plugin\_ctgan*), 47  
 CTGANPlugin.Config (class in *synthcity.plugins.generic.plugin\_ctgan*), 49  
 cuda() (*ConditionalDiscriminator* method), 476  
 cuda() (*ConditionalGenerator* method), 489  
 cuda() (*ConvNet* method), 502  
 cuda() (*Decoder* method), 277  
 cuda() (*DynamicDeepHitLayers* method), 446  
 cuda() (*Encoder* method), 289  
 cuda() (*GAN* method), 249  
 cuda() (*ImageGAN* method), 518  
 cuda() (*LatentODE* method), 388  
 cuda() (*LinearLayer* method), 221  
 cuda() (*MLP* method), 234  
 cuda() (*NormalizingFlows* method), 264  
 cuda() (*RadialGAN* method), 93  
 cuda() (*TabularFlows* method), 360  
 cuda() (*TabularGAN* method), 332  
 cuda() (*TabularRadialGAN* method), 111  
 cuda() (*TabularVAE* method), 346  
 cuda() (*TimeEventGAN* method), 462  
 cuda() (*TimeSeriesDecoder* method), 401  
 cuda() (*TimeSeriesEncoder* method), 414  
 cuda() (*TimeSeriesGAN* method), 375  
 cuda() (*TimeSeriesVAE* method), 429  
 cuda() (*TransformerModel* method), 316  
 cuda() (*VAE* method), 303  
 D  
 data (*CategoricalDistribution* attribute), 200  
 data (*DatetimeDistribution* attribute), 202  
 data (*Distribution* attribute), 204  
 data (*FloatDistribution* attribute), 206  
 data (*IntegerDistribution* attribute), 209  
 data (*IntLogDistribution* attribute), 208  
 data (*LogDistribution* attribute), 211  
 data (*Schema* attribute), 214  
 dataframe() (*DataLoader* method), 30  
 dataframe() (*GenericDataLoader* method), 31  
 dataframe() (*ImageDataLoader* method), 33  
 dataframe() (*SurvivalAnalysisDataLoader* method), 34  
 dataframe() (*TimeSeriesDataLoader* method), 36  
 dataframe() (*TimeSeriesSurvivalDataLoader* method), 38  
 DataLeakageLinear (class in *synthcity.metrics.eval\_attacks*), 187  
 DataLeakageMLP (class in *synthcity.metrics.eval\_attacks*), 188  
 DataLeakageXGB (class in *synthcity.metrics.eval\_attacks*), 188  
 DataLoader (class in *synthcity.plugins.core.data\_loader*), 29  
 dataloader() (*GAN* method), 250



- dataloader() (*ImageGAN method*), 518  
 dataloader() (*NormalizingFlows method*), 264  
 dataloader() (*RadialGAN method*), 93  
 dataloader() (*TimeEventGAN method*), 462  
 dataloader() (*TimeSeriesGAN method*), 376  
 DataMismatchScore (class in *synthcity.metrics.eval\_sanity*), 166  
 DatetimeDistribution (class in *synthcity.plugins.core.distribution*), 201  
 DatetimeDistribution.Config (class in *synthcity.plugins.core.distribution*), 201  
 DATETIMEToEvent (class in *synthcity.plugins.core.models.time\_to\_event.tte\_date*), 459  
 DECAFPlugin (class in *synthcity.plugins.privacy.plugin\_decaf*), 85  
 DECAFPlugin.Config (class in *synthcity.plugins.privacy.plugin\_decaf*), 87  
 decode() (*DataLoader method*), 30  
 decode() (*GenericDataLoader method*), 31  
 decode() (*ImageDataLoader method*), 33  
 decode() (*SurvivalAnalysisDataLoader method*), 34  
 decode() (*TabularFlows method*), 361  
 decode() (*TabularGAN method*), 332  
 decode() (*TabularRadialGAN method*), 111  
 decode() (*TabularVAE method*), 346  
 decode() (*TimeSeriesDataLoader method*), 36  
 decode() (*TimeSeriesSurvivalDataLoader method*), 38  
 Decoder (class in *synthcity.plugins.core.models.vae*), 275  
 decompress() (*DataLoader method*), 30  
 decompress() (*GenericDataLoader method*), 31  
 decompress() (*ImageDataLoader method*), 33  
 decompress() (*SurvivalAnalysisDataLoader method*), 34  
 decompress() (*TimeSeriesDataLoader method*), 36  
 decompress() (*TimeSeriesSurvivalDataLoader method*), 38  
 decorate() (*DataLoader method*), 30  
 decorate() (*GenericDataLoader method*), 31  
 decorate() (*ImageDataLoader method*), 33  
 decorate() (*SurvivalAnalysisDataLoader method*), 34  
 decorate() (*TimeSeriesDataLoader method*), 36  
 decorate() (*TimeSeriesSurvivalDataLoader method*), 38  
 DeepHitSurvivalAnalysis (class in *synthcity.plugins.core.models.survival\_analysis.surv\_deephit*), 442  
 DeltaPresence (class in *synthcity.metrics.eval\_privacy*), 181  
 DetectionEvaluator (class in *synthcity.metrics.eval\_detection*), 178  
 dict() (*CategoricalDistribution method*), 200  
 dict() (*Constraints method*), 198  
 dict() (*DatetimeDistribution method*), 202  
 dict() (*Distribution method*), 204  
 dict() (*FeatureInfo method*), 216  
 dict() (*FloatDistribution method*), 206  
 dict() (*IntegerDistribution method*), 210  
 dict() (*IntLogDistribution method*), 208  
 dict() (*LogDistribution method*), 211  
 dict() (*Schema method*), 214  
 direction() (*AlphaPrecision static method*), 168  
 direction() (*AttackEvaluator static method*), 187  
 direction() (*AugmentationPerformanceEvaluatorLinear static method*), 174  
 direction() (*AugmentationPerformanceEvaluatorMLP static method*), 175  
 direction() (*AugmentationPerformanceEvaluatorXGB static method*), 175  
 direction() (*BasicMetricEvaluator static method*), 165  
 direction() (*ChiSquaredTest static method*), 169  
 direction() (*CloseValuesProbability static method*), 166  
 direction() (*CommonRowsProportion static method*), 166  
 direction() (*DataLeakageLinear static method*), 187  
 direction() (*DataLeakageMLP static method*), 188  
 direction() (*DataLeakageXGB static method*), 188  
 direction() (*DataMismatchScore static method*), 167  
 direction() (*DeltaPresence static method*), 181  
 direction() (*DetectionEvaluator static method*), 179  
 direction() (*DistantValuesProbability static method*), 167  
 direction() (*DomiasMIA static method*), 182  
 direction() (*DomiasMIABNAF static method*), 182  
 direction() (*DomiasMIKDE static method*), 183  
 direction() (*DomiasMIAPrior static method*), 183  
 direction() (*FeatureImportanceRankDistance static method*), 176  
 direction() (*FrechetInceptionDistance static method*), 170  
 direction() (*IdentifiabilityScore static method*), 184  
 direction() (*InverseKLDivergence static method*), 170  
 direction() (*JensenShannonDistance static method*), 171  
 direction() (*kAnonymization static method*), 185  
 direction() (*kMap static method*), 186  
 direction() (*KolmogorovSmirnovTest static method*), 171  
 direction() (*IDiversityDistinct static method*), 186  
 direction() (*MaximumMeanDiscrepancy static method*), 172  
 direction() (*NearestSyntheticNeighborDistance static method*), 168  
 direction() (*PerformanceEvaluator static method*), 176  
 direction() (*PerformanceEvaluatorLinear static method*), 177

- `direction()` (*PerformanceEvaluatorMLP static method*), 177
- `direction()` (*PerformanceEvaluatorXGB static method*), 178
- `direction()` (*PRDCScore static method*), 172
- `direction()` (*PrivacyEvaluator static method*), 184
- `direction()` (*StatisticalEvaluator static method*), 173
- `direction()` (*SurvivalKMDistance static method*), 173
- `direction()` (*SyntheticDetectionGMM static method*), 179
- `direction()` (*SyntheticDetectionLinear static method*), 180
- `direction()` (*SyntheticDetectionMLP static method*), 180
- `direction()` (*SyntheticDetectionXGB static method*), 181
- `direction()` (*WassersteinDistance static method*), 174
- `direction()` (*WeightedMetrics method*), 189
- `discretize()` (*DynamicDeepHitModel method*), 457
- `display_imgs()` (in module *synthcity.plugins.core.models.image\_gan*), 529
- `display_network()` (*PrivBayes method*), 76
- `distance()` (*FeatureImportanceRankDistance method*), 176
- `DistantValuesProbability` (class in *synthcity.metrics.eval\_sanity*), 167
- `Distribution` (class in *synthcity.plugins.core.distribution*), 203
- `Distribution.Config` (class in *synthcity.plugins.core.distribution*), 204
- `domain` (*Schema attribute*), 214
- `domain()` (*DataLoader method*), 30
- `domain()` (*GenericDataLoader method*), 31
- `domain()` (*ImageDataLoader method*), 33
- `domain()` (*SurvivalAnalysisDataLoader method*), 35
- `domain()` (*TimeSeriesDataLoader method*), 36
- `domain()` (*TimeSeriesSurvivalDataLoader method*), 38
- `DomiasMIA` (class in *synthcity.metrics.eval\_privacy*), 182
- `DomiasMIABNAF` (class in *synthcity.metrics.eval\_privacy*), 182
- `DomiasMIAKDE` (class in *synthcity.metrics.eval\_privacy*), 183
- `DomiasMIAPrior` (class in *synthcity.metrics.eval\_privacy*), 183
- `double()` (*ConditionalDiscriminator method*), 476
- `double()` (*ConditionalGenerator method*), 489
- `double()` (*ConvNet method*), 502
- `double()` (*Decoder method*), 277
- `double()` (*DynamicDeepHitLayers method*), 446
- `double()` (*Encoder method*), 290
- `double()` (*GAN method*), 250
- `double()` (*ImageGAN method*), 518
- `double()` (*LatentODE method*), 389
- `double()` (*LinearLayer method*), 221
- `double()` (*MLP method*), 235
- `double()` (*NormalizingFlows method*), 264
- `double()` (*RadialGAN method*), 93
- `double()` (*TabularFlows method*), 361
- `double()` (*TabularGAN method*), 332
- `double()` (*TabularRadialGAN method*), 112
- `double()` (*TabularVAE method*), 346
- `double()` (*TimeEventGAN method*), 462
- `double()` (*TimeSeriesDecoder method*), 402
- `double()` (*TimeSeriesEncoder method*), 415
- `double()` (*TimeSeriesGAN method*), 376
- `double()` (*TimeSeriesVAE method*), 430
- `double()` (*TransformerModel method*), 317
- `double()` (*VAE method*), 304
- `DPGANPlugin` (class in *synthcity.plugins.privacy.plugin\_dpgan*), 81
- `DPGANPlugin.Config` (class in *synthcity.plugins.privacy.plugin\_dpgan*), 82
- `drop()` (*DataLoader method*), 30
- `drop()` (*GenericDataLoader method*), 31
- `drop()` (*ImageDataLoader method*), 33
- `drop()` (*SurvivalAnalysisDataLoader method*), 35
- `drop()` (*TimeSeriesDataLoader method*), 36
- `drop()` (*TimeSeriesSurvivalDataLoader method*), 38
- `dtype()` (*CategoricalDistribution method*), 200
- `dtype()` (*DatetimeDistribution method*), 202
- `dtype()` (*Distribution method*), 204
- `dtype()` (*FloatDistribution method*), 206
- `dtype()` (*IntegerDistribution method*), 210
- `dtype()` (*IntLogDistribution method*), 208
- `dtype()` (*LogDistribution method*), 211
- `dump_patches` (*ConditionalDiscriminator attribute*), 476
- `dump_patches` (*ConditionalGenerator attribute*), 489
- `dump_patches` (*ConvNet attribute*), 502
- `dump_patches` (*Decoder attribute*), 277
- `dump_patches` (*DynamicDeepHitLayers attribute*), 446
- `dump_patches` (*Encoder attribute*), 290
- `dump_patches` (*GAN attribute*), 250
- `dump_patches` (*ImageGAN attribute*), 519
- `dump_patches` (*LatentODE attribute*), 389
- `dump_patches` (*LinearLayer attribute*), 221
- `dump_patches` (*MLP attribute*), 235
- `dump_patches` (*NormalizingFlows attribute*), 264
- `dump_patches` (*RadialGAN attribute*), 94
- `dump_patches` (*TabularFlows attribute*), 361
- `dump_patches` (*TabularGAN attribute*), 332
- `dump_patches` (*TabularRadialGAN attribute*), 112
- `dump_patches` (*TabularVAE attribute*), 347
- `dump_patches` (*TimeEventGAN attribute*), 462
- `dump_patches` (*TimeSeriesDecoder attribute*), 402
- `dump_patches` (*TimeSeriesEncoder attribute*), 415
- `dump_patches` (*TimeSeriesGAN attribute*), 376
- `dump_patches` (*TimeSeriesVAE attribute*), 430

- [dump\\_patches \(TransformerModel attribute\), 317](#)  
[dump\\_patches \(VAE attribute\), 304](#)  
[DynamicDeepHitLayers \(class in synthcity.plugins.core.models.time\\_series\\_survival.ts\\_surv\\_dynamic\\_deep\\_hit\), 444](#)  
[DynamicDeepHitModel \(class in synthcity.plugins.core.models.time\\_series\\_survival.ts\\_surv\\_dynamic\\_deep\\_hit\), 457](#)  
[DynamicDeephitTimeSeriesSurvival \(class in synthcity.plugins.core.models.time\\_series\\_survival.ts\\_surv\\_dynamic\\_deep\\_hit\), 458](#)
- ## E
- [encode\(\) \(DataLoader method\), 30](#)  
[encode\(\) \(GenericDataLoader method\), 32](#)  
[encode\(\) \(ImageDataLoader method\), 33](#)  
[encode\(\) \(SurvivalAnalysisDataLoader method\), 35](#)  
[encode\(\) \(TabularFlows method\), 361](#)  
[encode\(\) \(TabularGAN method\), 332](#)  
[encode\(\) \(TabularRadialGAN method\), 112](#)  
[encode\(\) \(TabularVAE method\), 347](#)  
[encode\(\) \(TimeSeriesDataLoader method\), 36](#)  
[encode\(\) \(TimeSeriesSurvivalDataLoader method\), 38](#)  
[Encoder \(class in synthcity.plugins.core.models.vae\), 288](#)  
[eval\(\) \(ConditionalDiscriminator method\), 476](#)  
[eval\(\) \(ConditionalGenerator method\), 489](#)  
[eval\(\) \(ConvNet method\), 502](#)  
[eval\(\) \(Decoder method\), 277](#)  
[eval\(\) \(DynamicDeepHitLayers method\), 446](#)  
[eval\(\) \(Encoder method\), 290](#)  
[eval\(\) \(GAN method\), 250](#)  
[eval\(\) \(ImageGAN method\), 519](#)  
[eval\(\) \(LatentODE method\), 389](#)  
[eval\(\) \(LinearLayer method\), 221](#)  
[eval\(\) \(MLP method\), 235](#)  
[eval\(\) \(NormalizingFlows method\), 264](#)  
[eval\(\) \(RadialGAN method\), 94](#)  
[eval\(\) \(TabularFlows method\), 361](#)  
[eval\(\) \(TabularGAN method\), 332](#)  
[eval\(\) \(TabularRadialGAN method\), 112](#)  
[eval\(\) \(TabularVAE method\), 347](#)  
[eval\(\) \(TimeEventGAN method\), 462](#)  
[eval\(\) \(TimeSeriesDecoder method\), 402](#)  
[eval\(\) \(TimeSeriesEncoder method\), 415](#)  
[eval\(\) \(TimeSeriesGAN method\), 376](#)  
[eval\(\) \(TimeSeriesVAE method\), 430](#)  
[eval\(\) \(TransformerModel method\), 317](#)  
[eval\(\) \(VAE method\), 304](#)  
[evaluate\(\) \(AlphaPrecision method\), 168](#)  
[evaluate\(\) \(AttackEvaluator method\), 187](#)  
[evaluate\(\) \(AugmentationPerformanceEvaluatorLinear method\), 174](#)  
[evaluate\(\) \(AugmentationPerformanceEvaluatorMLP method\), 175](#)  
[evaluate\(\) \(AugmentationPerformanceEvaluatorXGB method\), 175](#)  
[evaluate\(\) \(BasicMetricEvaluator method\), 165](#)  
[evaluate\(\) \(Benchmarks static method\), 189](#)  
[evaluate\(\) \(CepiSquareTest method\), 169](#)  
[evaluate\(\) \(CloseValuesProbability method\), 166](#)  
[evaluate\(\) \(CommonRowsProportion method\), 166](#)  
[evaluate\(\) \(DataLeakageLinear method\), 187](#)  
[evaluate\(\) \(DataLeakageMLP method\), 188](#)  
[evaluate\(\) \(DataLeakageXGB method\), 188](#)  
[evaluate\(\) \(DataMismatchScore method\), 167](#)  
[evaluate\(\) \(DeltaPresence method\), 181](#)  
[evaluate\(\) \(DetectionEvaluator method\), 179](#)  
[evaluate\(\) \(DistantValuesProbability method\), 167](#)  
[evaluate\(\) \(DomiasMIA method\), 182](#)  
[evaluate\(\) \(DomiasMIABNAF method\), 182](#)  
[evaluate\(\) \(DomiasMIKDE method\), 183](#)  
[evaluate\(\) \(DomiasMIAPrior method\), 183](#)  
[evaluate\(\) \(FeatureImportanceRankDistance method\), 176](#)  
[evaluate\(\) \(FrechetInceptionDistance method\), 170](#)  
[evaluate\(\) \(IdentifiabilityScore method\), 184](#)  
[evaluate\(\) \(InverseKLDivergence method\), 170](#)  
[evaluate\(\) \(JensenShannonDistance method\), 171](#)  
[evaluate\(\) \(kAnonymization method\), 185](#)  
[evaluate\(\) \(kMap method\), 186](#)  
[evaluate\(\) \(KolmogorovSmirnovTest method\), 171](#)  
[evaluate\(\) \(IDiversityDistinct method\), 186](#)  
[evaluate\(\) \(MaximumMeanDiscrepancy method\), 172](#)  
[evaluate\(\) \(NearestSyntheticNeighborDistance method\), 168](#)  
[evaluate\(\) \(PerformanceEvaluator method\), 176](#)  
[evaluate\(\) \(PerformanceEvaluatorLinear method\), 177](#)  
[evaluate\(\) \(PerformanceEvaluatorMLP method\), 177](#)  
[evaluate\(\) \(PerformanceEvaluatorXGB method\), 178](#)  
[evaluate\(\) \(PRDCScore method\), 172](#)  
[evaluate\(\) \(PrivacyEvaluator method\), 184](#)  
[evaluate\(\) \(StatisticalEvaluator method\), 173](#)  
[evaluate\(\) \(SurvivalKMDistance method\), 173](#)  
[evaluate\(\) \(SyntheticDetectionGMM method\), 179](#)  
[evaluate\(\) \(SyntheticDetectionLinear method\), 180](#)  
[evaluate\(\) \(SyntheticDetectionMLP method\), 180](#)  
[evaluate\(\) \(SyntheticDetectionXGB method\), 181](#)  
[evaluate\(\) \(WassersteinDistance method\), 174](#)  
[evaluate\(\) \(WeightedMetrics method\), 189](#)  
[evaluate\\_data\(\) \(kAnonymization method\), 185](#)  
[evaluate\\_data\(\) \(IDiversityDistinct method\), 186](#)  
[evaluate\\_default\(\) \(AlphaPrecision method\), 168](#)  
[evaluate\\_default\(\) \(AttackEvaluator method\), 187](#)  
[evaluate\\_default\(\) \(AugmentationPerformanceEvaluatorLinear method\), 174](#)

- `evaluate_default()` (*AugmentationPerformanceEvaluatorMLP method*), 175
- `evaluate_default()` (*AugmentationPerformanceEvaluatorXGB method*), 175
- `evaluate_default()` (*BasicMetricEvaluator method*), 165
- `evaluate_default()` (*ChiSquaredTest method*), 169
- `evaluate_default()` (*CloseValuesProbability method*), 166
- `evaluate_default()` (*CommonRowsProportion method*), 166
- `evaluate_default()` (*DataLeakageLinear method*), 187
- `evaluate_default()` (*DataLeakageMLP method*), 188
- `evaluate_default()` (*DataLeakageXGB method*), 188
- `evaluate_default()` (*DataMismatchScore method*), 167
- `evaluate_default()` (*DeltaPresence method*), 182
- `evaluate_default()` (*DetectionEvaluator method*), 179
- `evaluate_default()` (*DistantValuesProbability method*), 167
- `evaluate_default()` (*DomiasMIA method*), 182
- `evaluate_default()` (*DomiasMIABNAF method*), 182
- `evaluate_default()` (*DomiasMIAKDE method*), 183
- `evaluate_default()` (*DomiasMIAPrior method*), 183
- `evaluate_default()` (*FeatureImportanceRankDistance method*), 176
- `evaluate_default()` (*FrechetInceptionDistance method*), 170
- `evaluate_default()` (*IdentifiabilityScore method*), 184
- `evaluate_default()` (*InverseKLDivergence method*), 170
- `evaluate_default()` (*JensenShannonDistance method*), 171
- `evaluate_default()` (*kAnonymization method*), 185
- `evaluate_default()` (*kMap method*), 186
- `evaluate_default()` (*KolmogorovSmirnovTest method*), 171
- `evaluate_default()` (*IDiversityDistinct method*), 186
- `evaluate_default()` (*MaximumMeanDiscrepancy method*), 172
- `evaluate_default()` (*NearestSyntheticNeighborDistance method*), 168
- `evaluate_default()` (*PerformanceEvaluator method*), 176
- `evaluate_default()` (*PerformanceEvaluatorLinear method*), 177
- `evaluate_default()` (*PerformanceEvaluatorMLP method*), 177
- `evaluate_default()` (*PerformanceEvaluatorXGB method*), 178
- `evaluate_default()` (*PRDCScore method*), 172
- `evaluate_default()` (*PrivacyEvaluator method*), 185
- `evaluate_default()` (*StatisticalEvaluator method*), 173
- `evaluate_default()` (*SurvivalKMDistance method*), 173
- `evaluate_default()` (*SyntheticDetectionGMM method*), 179
- `evaluate_default()` (*SyntheticDetectionLinear method*), 180
- `evaluate_default()` (*SyntheticDetectionMLP method*), 180
- `evaluate_default()` (*SyntheticDetectionXGB method*), 181
- `evaluate_default()` (*WassersteinDistance method*), 174
- `evaluate_p_R()` (*DomiasMIA method*), 182
- `evaluate_p_R()` (*DomiasMIABNAF method*), 183
- `evaluate_p_R()` (*DomiasMIAKDE method*), 183
- `evaluate_p_R()` (*DomiasMIAPrior method*), 183
- `explain()` (*XGBSurvivalAnalysis method*), 443
- `extend()` (*Constraints method*), 198
- `extra_repr()` (*ConditionalDiscriminator method*), 476
- `extra_repr()` (*ConditionalGenerator method*), 489
- `extra_repr()` (*ConvNet method*), 503
- `extra_repr()` (*Decoder method*), 277
- `extra_repr()` (*DynamicDeepHitLayers method*), 446
- `extra_repr()` (*Encoder method*), 290
- `extra_repr()` (*GAN method*), 250
- `extra_repr()` (*ImageGAN method*), 519
- `extra_repr()` (*LatentODE method*), 389
- `extra_repr()` (*LinearLayer method*), 221
- `extra_repr()` (*MLP method*), 235
- `extra_repr()` (*NormalizingFlows method*), 264
- `extra_repr()` (*RadialGAN method*), 94
- `extra_repr()` (*TabularFlows method*), 361
- `extra_repr()` (*TabularGAN method*), 332
- `extra_repr()` (*TabularRadialGAN method*), 112
- `extra_repr()` (*TabularVAE method*), 347
- `extra_repr()` (*TimeEventGAN method*), 462
- `extra_repr()` (*TimeSeriesDecoder method*), 402
- `extra_repr()` (*TimeSeriesEncoder method*), 415
- `extra_repr()` (*TimeSeriesGAN method*), 376
- `extra_repr()` (*TimeSeriesVAE method*), 430
- `extra_repr()` (*TransformerModel method*), 317
- `extra_repr()` (*VAE method*), 304
- `extract_masked_features()` (*TimeSeriesDataLoader static method*), 36
- `extract_masked_features()` (*TimeSeriesSurvivalDataLoader static method*), 39

## F

- `feature` (*network\_edge attribute*), 80
- `feature_constraints()` (*Constraints method*), 198
- `feature_params()` (*Constraints method*), 198



- feature\_type (*FeatureInfo* attribute), 216
- FeatureImportanceRankDistance (class in *synthcity.metrics.eval\_performance*), 175
- FeatureInfo (class in *synthcity.plugins.core.models.tabular\_encoder*), 216
- features() (*Constraints* method), 198
- features() (*Schema* method), 214
- fillna() (*DataLoader* method), 30
- fillna() (*GenericDataLoader* method), 32
- fillna() (*ImageDataLoader* method), 33
- fillna() (*SurvivalAnalysisDataLoader* method), 35
- fillna() (*TimeSeriesDataLoader* method), 36
- fillna() (*TimeSeriesSurvivalDataLoader* method), 39
- filter() (*Constraints* method), 198
- filter\_ids() (*TimeSeriesDataLoader* method), 36
- filter\_ids() (*TimeSeriesSurvivalDataLoader* method), 39
- filter\_indices() (*FlexibleDataset* method), 40
- fit() (*AdsGANPlugin* method), 67
- fit() (*ARFPlugin* method), 25
- fit() (*BayesianNetworkPlugin* method), 44
- fit() (*BinEncoder* method), 215
- fit() (*ConvNet* method), 503
- fit() (*CoxPHSurvivalAnalysis* method), 441
- fit() (*CoxTimeSeriesSurvival* method), 443
- fit() (*CTGANPlugin* method), 49
- fit() (*DATETimeToEvent* method), 459
- fit() (*DECAFPPlugin* method), 87
- fit() (*DeepHitSurvivalAnalysis* method), 442
- fit() (*DPGANPlugin* method), 83
- fit() (*DynamicDeepHitModel* method), 457
- fit() (*DynamicDeepHitTimeSeriesSurvival* method), 458
- fit() (*FourierFlowsPlugin* method), 146
- fit() (*GAN* method), 250
- fit() (*GOGGLEPlugin* method), 22
- fit() (*ImageAdsGANPlugin* method), 160
- fit() (*ImageCGANPlugin* method), 155
- fit() (*ImageGAN* method), 519
- fit() (*MLP* method), 235
- fit() (*NormalizingFlows* method), 264
- fit() (*NormalizingFlowsPlugin* method), 54
- fit() (*PATEGAN* method), 70
- fit() (*PATEGANPlugin* method), 72
- fit() (*Plugin* method), 194
- fit() (*PrivBayes* method), 76
- fit() (*PrivBayesPlugin* method), 77
- fit() (*RadialGAN* method), 94
- fit() (*RadialGANPlugin* method), 106
- fit() (*RTVAEPlugin* method), 58
- fit() (*SurVAEPlugin* method), 132
- fit() (*SurvivalCTGANPlugin* method), 128
- fit() (*SurvivalFunctionTimeToEvent* method), 473
- fit() (*SurvivalGANPlugin* method), 124
- fit() (*SurvivalNFlowPlugin* method), 136
- fit() (*TabularEncoder* method), 217
- fit() (*TabularFlows* method), 361
- fit() (*TabularGAN* method), 333
- fit() (*TabularRadialGAN* method), 112
- fit() (*TabularVAE* method), 347
- fit() (*Teachers* method), 75
- fit() (*TimeEventGAN* method), 463
- fit() (*TimeGANPlugin* method), 141
- fit() (*TimeSeriesBinEncoder* method), 218
- fit() (*TimeSeriesGAN* method), 376
- fit() (*TimeSeriesTabularEncoder* method), 218
- fit() (*TimeSeriesVAE* method), 430
- fit() (*TimeVAEPlugin* method), 151
- fit() (*TVAEPlugin* method), 62
- fit() (*VAE* method), 304
- fit() (*WeibullAFTSurvivalAnalysis* method), 441
- fit() (*XGBSurvivalAnalysis* method), 443
- fit() (*XGBTimeSeriesSurvival* method), 459
- fit\_temporal() (*TimeSeriesTabularEncoder* method), 218
- fit\_transform() (*TimeSeriesBinEncoder* method), 218
- fit\_transform() (*TimeSeriesTabularEncoder* method), 218
- fit\_transform\_temporal() (*TimeSeriesTabularEncoder* method), 219
- FlexibleDataset (class in *synthcity.plugins.core.dataset*), 40
- float() (*ConditionalDiscriminator* method), 477
- float() (*ConditionalGenerator* method), 489
- float() (*ConvNet* method), 503
- float() (*Decoder* method), 277
- float() (*DynamicDeepHitLayers* method), 446
- float() (*Encoder* method), 290
- float() (*GAN* method), 250
- float() (*ImageGAN* method), 519
- float() (*LatentODE* method), 389
- float() (*LinearLayer* method), 221
- float() (*MLP* method), 235
- float() (*NormalizingFlows* method), 264
- float() (*RadialGAN* method), 94
- float() (*TabularFlows* method), 361
- float() (*TabularGAN* method), 333
- float() (*TabularRadialGAN* method), 112
- float() (*TabularVAE* method), 347
- float() (*TimeEventGAN* method), 463
- float() (*TimeSeriesDecoder* method), 402
- float() (*TimeSeriesEncoder* method), 415
- float() (*TimeSeriesGAN* method), 376
- float() (*TimeSeriesVAE* method), 430
- float() (*TransformerModel* method), 317
- float() (*VAE* method), 304

- FloatDistribution (class in synthcity.plugins.core.distribution), 205
- FloatDistribution.Config (class in synthcity.plugins.core.distribution), 205
- forward() (ConditionalDiscriminator method), 477
- forward() (ConditionalGenerator method), 490
- forward() (ConvNet method), 503
- forward() (Decoder method), 278
- forward() (DynamicDeepHitLayers method), 447
- forward() (Encoder method), 290
- forward() (GAN method), 251
- forward() (ImageGAN method), 519
- forward() (LatentODE method), 389
- forward() (LinearLayer method), 222
- forward() (MLP method), 235
- forward() (NormalizingFlows method), 265
- forward() (RadialGAN method), 94
- forward() (TabularFlows method), 361
- forward() (TabularGAN method), 333
- forward() (TabularRadialGAN method), 112
- forward() (TabularVAE method), 347
- forward() (TimeEventGAN method), 463
- forward() (TimeSeriesDecoder method), 402
- forward() (TimeSeriesEncoder method), 415
- forward() (TimeSeriesGAN method), 376
- forward() (TimeSeriesVAE method), 430
- forward() (TransformerModel method), 317
- forward() (VAE method), 305
- forward\_attention() (DynamicDeepHitLayers method), 447
- forward\_emb() (DynamicDeepHitLayers method), 447
- FourierFlowsPlugin (class in synthcity.plugins.time\_series.plugin\_fflows), 144
- FourierFlowsPlugin.Config (class in synthcity.plugins.time\_series.plugin\_fflows), 146
- fqdn() (AdsGANPlugin class method), 68
- fqdn() (AlphaPrecision class method), 169
- fqdn() (ARFPlugin class method), 26
- fqdn() (AttackEvaluator class method), 187
- fqdn() (AugmentationPerformanceEvaluatorLinear class method), 174
- fqdn() (AugmentationPerformanceEvaluatorMLP class method), 175
- fqdn() (AugmentationPerformanceEvaluatorXGB class method), 175
- fqdn() (BasicMetricEvaluator class method), 165
- fqdn() (BayesianNetworkPlugin class method), 45
- fqdn() (ChiSquaredTest class method), 169
- fqdn() (CloseValuesProbability class method), 166
- fqdn() (CommonRowsProportion class method), 166
- fqdn() (CTGANPlugin class method), 50
- fqdn() (DataLeakageLinear class method), 187
- fqdn() (DataLeakageMLP class method), 188
- fqdn() (DataLeakageXGB class method), 188
- fqdn() (DataMismatchScore class method), 167
- fqdn() (DECAFFPlugin class method), 88
- fqdn() (DeltaPresence class method), 182
- fqdn() (DetectionEvaluator class method), 179
- fqdn() (DistantValuesProbability class method), 167
- fqdn() (DomiasMIA class method), 182
- fqdn() (DomiasMIABNAF class method), 183
- fqdn() (DomiasMIAKDE class method), 183
- fqdn() (DomiasMIAPrior class method), 183
- fqdn() (DPGANPlugin class method), 83
- fqdn() (FeatureImportanceRankDistance class method), 176
- fqdn() (FourierFlowsPlugin class method), 147
- fqdn() (FrechetInceptionDistance class method), 170
- fqdn() (GOGGLEPlugin class method), 22
- fqdn() (IdentifiabilityScore class method), 184
- fqdn() (ImageAdsGANPlugin class method), 161
- fqdn() (ImageCGANPlugin class method), 156
- fqdn() (InverseKLDivergence class method), 170
- fqdn() (JensenShannonDistance class method), 171
- fqdn() (kAnonymization class method), 185
- fqdn() (kMap class method), 186
- fqdn() (KolmogorovSmirnovTest class method), 171
- fqdn() (lDiversityDistinct class method), 186
- fqdn() (MaximumMeanDiscrepancy class method), 172
- fqdn() (NearestSyntheticNeighborDistance class method), 168
- fqdn() (NormalizingFlowsPlugin class method), 55
- fqdn() (PATEGANPlugin class method), 73
- fqdn() (PerformanceEvaluator class method), 176
- fqdn() (PerformanceEvaluatorLinear class method), 177
- fqdn() (PerformanceEvaluatorMLP class method), 177
- fqdn() (PerformanceEvaluatorXGB class method), 178
- fqdn() (Plugin class method), 195
- fqdn() (PRDCScore class method), 172
- fqdn() (PrivacyEvaluator class method), 185
- fqdn() (PrivBayesPlugin class method), 78
- fqdn() (RadialGANPlugin class method), 107
- fqdn() (RTVAEPlugin class method), 59
- fqdn() (StatisticalEvaluator class method), 173
- fqdn() (SurVAEPlugin class method), 133
- fqdn() (SurvivalCTGANPlugin class method), 129
- fqdn() (SurvivalGANPlugin class method), 125
- fqdn() (SurvivalKMDistance class method), 173
- fqdn() (SurvivalNFlowPlugin class method), 137
- fqdn() (SyntheticDetectionGMM class method), 179
- fqdn() (SyntheticDetectionLinear class method), 180
- fqdn() (SyntheticDetectionMLP class method), 180
- fqdn() (SyntheticDetectionXGB class method), 181
- fqdn() (TimeGANPlugin class method), 142
- fqdn() (TimeVAEPlugin class method), 152
- fqdn() (TVAEPlugin class method), 63
- fqdn() (WassersteinDistance class method), 174

- FrechetInceptionDistance (class in synthcity.metrics.eval\_statistical), 169
- from\_constraints() (Schema class method), 214
- from\_info() (DataLoader static method), 30
- from\_info() (GenericDataLoader static method), 32
- from\_info() (ImageDataLoader static method), 33
- from\_info() (SurvivalAnalysisDataLoader static method), 35
- from\_info() (TimeSeriesDataLoader static method), 36
- from\_info() (TimeSeriesSurvivalDataLoader static method), 39
- from\_orm() (CategoricalDistribution class method), 200
- from\_orm() (Constraints class method), 198
- from\_orm() (DatetimeDistribution class method), 202
- from\_orm() (Distribution class method), 204
- from\_orm() (FeatureInfo class method), 216
- from\_orm() (FloatDistribution class method), 206
- from\_orm() (IntegerDistribution class method), 210
- from\_orm() (IntLogDistribution class method), 208
- from\_orm() (LogDistribution class method), 211
- from\_orm() (Schema class method), 214
- ## G
- GAN (class in synthcity.plugins.core.models.gan), 246
- generate() (AdsGANPlugin method), 68
- generate() (ARFPlugin method), 26
- generate() (BayesianNetworkPlugin method), 45
- generate() (CTGANPlugin method), 50
- generate() (DECAFPlugin method), 88
- generate() (DPGANPlugin method), 84
- generate() (FourierFlowsPlugin method), 147
- generate() (GAN method), 251
- generate() (GOGGLEPlugin method), 22
- generate() (ImageAdsGANPlugin method), 161
- generate() (ImageCGANPlugin method), 156
- generate() (ImageGAN method), 519
- generate() (NormalizingFlows method), 265
- generate() (NormalizingFlowsPlugin method), 55
- generate() (PATEGANPlugin method), 73
- generate() (Plugin method), 195
- generate() (PrivBayesPlugin method), 78
- generate() (RadialGAN method), 94
- generate() (RadialGANPlugin method), 107
- generate() (RTVAEPlugin method), 59
- generate() (SurVAEPlugin method), 133
- generate() (SurvivalCTGANPlugin method), 129
- generate() (SurvivalGANPlugin method), 125
- generate() (SurvivalNFlowPlugin method), 137
- generate() (TabularFlows method), 362
- generate() (TabularGAN method), 333
- generate() (TabularRadialGAN method), 112
- generate() (TabularVAE method), 347
- generate() (TimeEventGAN method), 463
- generate() (TimeGANPlugin method), 142
- generate() (TimeSeriesGAN method), 377
- generate() (TimeSeriesVAE method), 431
- generate() (TimeVAEPlugin method), 152
- generate() (TVAEPlugin method), 63
- generate() (VAE method), 305
- GenericDataLoader (class in synthcity.plugins.core.dataloader), 30
- get() (CategoricalDistribution method), 200
- get() (DatetimeDistribution method), 202
- get() (Distribution method), 204
- get() (FloatDistribution method), 206
- get() (IntegerDistribution method), 210
- get() (IntLogDistribution method), 208
- get() (LogDistribution method), 211
- get() (PluginLoader method), 196
- get() (Schema method), 214
- get\_buffer() (ConditionalDiscriminator method), 477
- get\_buffer() (ConditionalGenerator method), 490
- get\_buffer() (ConvNet method), 503
- get\_buffer() (Decoder method), 278
- get\_buffer() (DynamicDeepHitLayers method), 447
- get\_buffer() (Encoder method), 291
- get\_buffer() (GAN method), 251
- get\_buffer() (ImageGAN method), 519
- get\_buffer() (LatentODE method), 390
- get\_buffer() (LinearLayer method), 222
- get\_buffer() (MLP method), 236
- get\_buffer() (NormalizingFlows method), 265
- get\_buffer() (RadialGAN method), 94
- get\_buffer() (TabularFlows method), 362
- get\_buffer() (TabularGAN method), 333
- get\_buffer() (TabularRadialGAN method), 112
- get\_buffer() (TabularVAE method), 347
- get\_buffer() (TimeEventGAN method), 463
- get\_buffer() (TimeSeriesDecoder method), 403
- get\_buffer() (TimeSeriesEncoder method), 416
- get\_buffer() (TimeSeriesGAN method), 377
- get\_buffer() (TimeSeriesVAE method), 431
- get\_buffer() (TransformerModel method), 318
- get\_buffer() (VAE method), 305
- get\_column\_info() (BinEncoder method), 216
- get\_column\_info() (TabularEncoder method), 218
- get\_dag() (DECAFPlugin method), 89
- get\_encoder() (TabularFlows method), 362
- get\_encoder() (TabularGAN method), 333
- get\_encoder() (TabularVAE method), 348
- get\_extra\_state() (ConditionalDiscriminator method), 477
- get\_extra\_state() (ConditionalGenerator method), 490
- get\_extra\_state() (ConvNet method), 503
- get\_extra\_state() (Decoder method), 278
- get\_extra\_state() (DynamicDeepHitLayers method), 447



- `get_extra_state()` (*Encoder method*), 291
  - `get_extra_state()` (*GAN method*), 251
  - `get_extra_state()` (*ImageGAN method*), 520
  - `get_extra_state()` (*LatentODE method*), 390
  - `get_extra_state()` (*LinearLayer method*), 222
  - `get_extra_state()` (*MLP method*), 236
  - `get_extra_state()` (*NormalizingFlows method*), 265
  - `get_extra_state()` (*RadialGAN method*), 95
  - `get_extra_state()` (*TabularFlows method*), 362
  - `get_extra_state()` (*TabularGAN method*), 333
  - `get_extra_state()` (*TabularRadialGAN method*), 113
  - `get_extra_state()` (*TabularVAE method*), 348
  - `get_extra_state()` (*TimeEventGAN method*), 463
  - `get_extra_state()` (*TimeSeriesDecoder method*), 403
  - `get_extra_state()` (*TimeSeriesEncoder method*), 416
  - `get_extra_state()` (*TimeSeriesGAN method*), 377
  - `get_extra_state()` (*TimeSeriesVAE method*), 431
  - `get_extra_state()` (*TransformerModel method*), 318
  - `get_extra_state()` (*VAE method*), 305
  - `get_fairness_column()` (*DataLoader method*), 30
  - `get_fairness_column()` (*GenericDataLoader method*), 32
  - `get_fairness_column()` (*ImageDataLoader method*), 33
  - `get_fairness_column()` (*SurvivalAnalysisDataLoader method*), 35
  - `get_fairness_column()` (*TimeSeriesDataLoader method*), 36
  - `get_fairness_column()` (*TimeSeriesSurvivalDataLoader method*), 39
  - `get_json_serializable_kwargs()` (*in module synthcity.benchmark.utils*), 191
  - `get_parameter()` (*ConditionalDiscriminator method*), 477
  - `get_parameter()` (*ConditionalGenerator method*), 490
  - `get_parameter()` (*ConvNet method*), 504
  - `get_parameter()` (*Decoder method*), 278
  - `get_parameter()` (*DynamicDeepHitLayers method*), 447
  - `get_parameter()` (*Encoder method*), 291
  - `get_parameter()` (*GAN method*), 251
  - `get_parameter()` (*ImageGAN method*), 520
  - `get_parameter()` (*LatentODE method*), 390
  - `get_parameter()` (*LinearLayer method*), 222
  - `get_parameter()` (*MLP method*), 236
  - `get_parameter()` (*NormalizingFlows method*), 265
  - `get_parameter()` (*RadialGAN method*), 95
  - `get_parameter()` (*TabularFlows method*), 362
  - `get_parameter()` (*TabularGAN method*), 334
  - `get_parameter()` (*TabularRadialGAN method*), 113
  - `get_parameter()` (*TabularVAE method*), 348
  - `get_parameter()` (*TimeEventGAN method*), 463
  - `get_parameter()` (*TimeSeriesDecoder method*), 403
  - `get_parameter()` (*TimeSeriesEncoder method*), 416
  - `get_parameter()` (*TimeSeriesGAN method*), 377
  - `get_parameter()` (*TimeSeriesVAE method*), 431
  - `get_parameter()` (*TransformerModel method*), 318
  - `get_parameter()` (*VAE method*), 305
  - `get_submodule()` (*ConditionalDiscriminator method*), 478
  - `get_submodule()` (*ConditionalGenerator method*), 490
  - `get_submodule()` (*ConvNet method*), 504
  - `get_submodule()` (*Decoder method*), 278
  - `get_submodule()` (*DynamicDeepHitLayers method*), 447
  - `get_submodule()` (*Encoder method*), 291
  - `get_submodule()` (*GAN method*), 251
  - `get_submodule()` (*ImageGAN method*), 520
  - `get_submodule()` (*LatentODE method*), 390
  - `get_submodule()` (*LinearLayer method*), 222
  - `get_submodule()` (*MLP method*), 236
  - `get_submodule()` (*NormalizingFlows method*), 266
  - `get_submodule()` (*RadialGAN method*), 95
  - `get_submodule()` (*TabularFlows method*), 362
  - `get_submodule()` (*TabularGAN method*), 334
  - `get_submodule()` (*TabularRadialGAN method*), 113
  - `get_submodule()` (*TabularVAE method*), 348
  - `get_submodule()` (*TimeEventGAN method*), 464
  - `get_submodule()` (*TimeSeriesDecoder method*), 403
  - `get_submodule()` (*TimeSeriesEncoder method*), 416
  - `get_submodule()` (*TimeSeriesGAN method*), 377
  - `get_submodule()` (*TimeSeriesVAE method*), 431
  - `get_submodule()` (*TransformerModel method*), 318
  - `get_submodule()` (*VAE method*), 305
  - `get_type()` (*PluginLoader method*), 197
  - GOGGLEPlugin (class in *synthcity.plugins.generic.plugin\_goggle*), 21
  - GOGGLEPlugin.Config (class in *synthcity.plugins.generic.plugin\_goggle*), 21
- ## H
- `half()` (*ConditionalDiscriminator method*), 478
  - `half()` (*ConditionalGenerator method*), 491
  - `half()` (*ConvNet method*), 504
  - `half()` (*Decoder method*), 279
  - `half()` (*DynamicDeepHitLayers method*), 448
  - `half()` (*Encoder method*), 292
  - `half()` (*GAN method*), 252
  - `half()` (*ImageGAN method*), 521
  - `half()` (*LatentODE method*), 391
  - `half()` (*LinearLayer method*), 223
  - `half()` (*MLP method*), 237
  - `half()` (*NormalizingFlows method*), 266
  - `half()` (*RadialGAN method*), 96
  - `half()` (*TabularFlows method*), 363
  - `half()` (*TabularGAN method*), 334
  - `half()` (*TabularRadialGAN method*), 114
  - `half()` (*TabularVAE method*), 349

- `half()` (*TimeEventGAN* method), 464
  - `half()` (*TimeSeriesDecoder* method), 404
  - `half()` (*TimeSeriesEncoder* method), 417
  - `half()` (*TimeSeriesGAN* method), 378
  - `half()` (*TimeSeriesVAE* method), 432
  - `half()` (*TransformerModel* method), 319
  - `half()` (*VAE* method), 306
  - `has()` (*CategoricalDistribution* method), 200
  - `has()` (*DatetimeDistribution* method), 202
  - `has()` (*Distribution* method), 204
  - `has()` (*FloatDistribution* method), 206
  - `has()` (*IntegerDistribution* method), 210
  - `has()` (*IntLogDistribution* method), 208
  - `has()` (*LogDistribution* method), 211
  - `hash()` (*DataLoader* method), 30
  - `hash()` (*GenericDataLoader* method), 32
  - `hash()` (*ImageDataLoader* method), 33
  - `hash()` (*SurvivalAnalysisDataLoader* method), 35
  - `hash()` (*TimeSeriesDataLoader* method), 36
  - `hash()` (*TimeSeriesSurvivalDataLoader* method), 39
  - `high` (*DatetimeDistribution* attribute), 202
  - `high` (*FloatDistribution* attribute), 206
  - `high` (*IntegerDistribution* attribute), 210
  - `high` (*IntLogDistribution* attribute), 208
  - `high` (*LogDistribution* attribute), 212
  - `highlight()` (*Benchmarks* static method), 190
  - `hyperparameter_space()` (*AdsGANPlugin* static method), 69
  - `hyperparameter_space()` (*ARFPlugin* static method), 27
  - `hyperparameter_space()` (*BayesianNetworkPlugin* static method), 46
  - `hyperparameter_space()` (*CoxPHSurvivalAnalysis* static method), 441
  - `hyperparameter_space()` (*CoxTimeSeriesSurvival* static method), 443
  - `hyperparameter_space()` (*CTGANPlugin* static method), 51
  - `hyperparameter_space()` (*DATETimeToEvent* static method), 459
  - `hyperparameter_space()` (*DECAFPlugin* static method), 89
  - `hyperparameter_space()` (*DeephitSurvivalAnalysis* static method), 442
  - `hyperparameter_space()` (*DPGANPlugin* static method), 84
  - `hyperparameter_space()` (*DynamicDeephitTimeSeriesSurvival* static method), 458
  - `hyperparameter_space()` (*FourierFlowsPlugin* static method), 148
  - `hyperparameter_space()` (*GOGGLEPlugin* static method), 23
  - `hyperparameter_space()` (*ImageAdsGANPlugin* static method), 162
  - `hyperparameter_space()` (*ImageCGANPlugin* static method), 157
  - `hyperparameter_space()` (*NormalizingFlowsPlugin* static method), 56
  - `hyperparameter_space()` (*PATEGANPlugin* static method), 74
  - `hyperparameter_space()` (*Plugin* static method), 195
  - `hyperparameter_space()` (*PrivBayesPlugin* static method), 79
  - `hyperparameter_space()` (*RadialGANPlugin* static method), 108
  - `hyperparameter_space()` (*RTVAEPlugin* static method), 60
  - `hyperparameter_space()` (*SurVAEPlugin* static method), 134
  - `hyperparameter_space()` (*SurvivalCTGANPlugin* static method), 130
  - `hyperparameter_space()` (*SurvivalFunctionTimeToEvent* static method), 473
  - `hyperparameter_space()` (*SurvivalGANPlugin* static method), 126
  - `hyperparameter_space()` (*SurvivalNFlowPlugin* static method), 138
  - `hyperparameter_space()` (*TimeGANPlugin* static method), 143
  - `hyperparameter_space()` (*TimeVAEPlugin* static method), 153
  - `hyperparameter_space()` (*TVAEPlugin* static method), 64
  - `hyperparameter_space()` (*WeibullAFTSurvivalAnalysis* static method), 441
  - `hyperparameter_space()` (*XGBSurvivalAnalysis* static method), 443
  - `hyperparameter_space()` (*XGBTimeSeriesSurvival* static method), 459
- I**
- `IdentifiabilityScore` (class in *synthcity.metrics.eval\_privacy*), 184
  - `ids()` (*TimeSeriesDataLoader* method), 36
  - `ids()` (*TimeSeriesSurvivalDataLoader* method), 39
  - `ImageAdsGANPlugin` (class in *synthcity.plugins.images.plugin\_image\_adsgan*), 158
  - `ImageAdsGANPlugin.Config` (class in *synthcity.plugins.images.plugin\_image\_adsgan*), 160
  - `ImageCGANPlugin` (class in *synthcity.plugins.images.plugin\_image\_cgan*), 154
  - `ImageCGANPlugin.Config` (class in *synthcity.plugins.images.plugin\_image\_cgan*), 155
  - `ImageDataLoader` (class in *synthcity.plugins.core.data\_loader*), 32

- ImageGAN (class in synthcity.plugins.core.models.image\_gan), 515
- includes() (CategoricalDistribution method), 200
- includes() (DatetimeDistribution method), 202
- includes() (Distribution method), 204
- includes() (FloatDistribution method), 206
- includes() (IntegerDistribution method), 210
- includes() (IntLogDistribution method), 208
- includes() (LogDistribution method), 212
- includes() (Schema method), 214
- index() (network\_edge method), 80
- info() (DataLoader method), 30
- info() (GenericDataLoader method), 32
- info() (ImageDataLoader method), 33
- info() (SurvivalAnalysisDataLoader method), 35
- info() (TimeSeriesDataLoader method), 36
- info() (TimeSeriesSurvivalDataLoader method), 39
- IntegerDistribution (class in synthcity.plugins.core.distribution), 209
- IntegerDistribution.Config (class in synthcity.plugins.core.distribution), 209
- IntLogDistribution (class in synthcity.plugins.core.distribution), 207
- IntLogDistribution.Config (class in synthcity.plugins.core.distribution), 207
- inverse\_transform() (BinEncoder method), 216
- inverse\_transform() (TabularEncoder method), 218
- inverse\_transform() (TimeSeriesTabularEncoder method), 219
- inverse\_transform\_observation\_times() (TimeSeriesTabularEncoder method), 219
- inverse\_transform\_static() (TimeSeriesTabularEncoder method), 219
- inverse\_transform\_temporal() (TimeSeriesTabularEncoder method), 219
- InverseKLDivergence (class in synthcity.metrics.eval\_statistical), 170
- ipu() (ConditionalDiscriminator method), 478
- ipu() (ConditionalGenerator method), 491
- ipu() (ConvNet method), 505
- ipu() (Decoder method), 279
- ipu() (DynamicDeepHitLayers method), 448
- ipu() (Encoder method), 292
- ipu() (GAN method), 252
- ipu() (ImageGAN method), 521
- ipu() (LatentODE method), 391
- ipu() (LinearLayer method), 223
- ipu() (MLP method), 237
- ipu() (NormalizingFlows method), 266
- ipu() (RadialGAN method), 96
- ipu() (TabularFlows method), 363
- ipu() (TabularGAN method), 335
- ipu() (TabularRadialGAN method), 114
- ipu() (TabularVAE method), 349
- ipu() (TimeEventGAN method), 464
- ipu() (TimeSeriesDecoder method), 404
- ipu() (TimeSeriesEncoder method), 417
- ipu() (TimeSeriesGAN method), 378
- ipu() (TimeSeriesVAE method), 432
- ipu() (TransformerModel method), 319
- ipu() (VAE method), 306
- is\_tabular() (DataLoader method), 30
- is\_tabular() (GenericDataLoader method), 32
- is\_tabular() (ImageDataLoader method), 33
- is\_tabular() (SurvivalAnalysisDataLoader method), 35
- is\_tabular() (TimeSeriesDataLoader method), 36
- is\_tabular() (TimeSeriesSurvivalDataLoader method), 39
- is\_valid() (Constraints method), 198
- ## J
- JensenShannonDistance (class in synthcity.metrics.eval\_statistical), 171
- json() (CategoricalDistribution method), 200
- json() (Constraints method), 199
- json() (DatetimeDistribution method), 202
- json() (Distribution method), 204
- json() (FeatureInfo method), 217
- json() (FloatDistribution method), 206
- json() (IntegerDistribution method), 210
- json() (IntLogDistribution method), 208
- json() (LogDistribution method), 212
- json() (Schema method), 214
- ## K
- kAnonymization (class in synthcity.metrics.eval\_privacy), 185
- kMap (class in synthcity.metrics.eval\_privacy), 185
- KolmogorovSmirnovTest (class in synthcity.metrics.eval\_statistical), 171
- ## L
- labels() (FlexibleDataset method), 40
- labels() (TensorDataset method), 41
- LatentODE (class in synthcity.plugins.core.models.ts\_vae), 387
- layout() (BinEncoder method), 216
- layout() (TabularEncoder method), 218
- layout() (TimeSeriesTabularEncoder method), 219
- LDiversityDistinct (class in synthcity.metrics.eval\_privacy), 186
- LinearLayer (class in synthcity.plugins.core.models.mlp), 219
- list() (PluginLoader method), 197
- load() (AdsGANPlugin static method), 69
- load() (ARFPlugin static method), 27
- load() (BayesianNetworkPlugin static method), 46



- `load()` (*CoxPHSurvivalAnalysis static method*), 441
- `load()` (*CoxTimeSeriesSurvival static method*), 444
- `load()` (*CTGANPlugin static method*), 51
- `load()` (*DATETimeToEvent static method*), 460
- `load()` (*DECAFPlugin static method*), 89
- `load()` (*DeepHitSurvivalAnalysis static method*), 442
- `load()` (*DPGANPlugin static method*), 84
- `load()` (*DynamicDeepHitTimeSeriesSurvival static method*), 458
- `load()` (*FourierFlowsPlugin static method*), 148
- `load()` (*GOGGLEPlugin static method*), 23
- `load()` (*ImageAdsGANPlugin static method*), 162
- `load()` (*ImageCGANPlugin static method*), 157
- `load()` (*NormalizingFlowsPlugin static method*), 56
- `load()` (*PATEGAN static method*), 70
- `load()` (*PATEGANPlugin static method*), 74
- `load()` (*Plugin static method*), 196
- `load()` (*PluginLoader method*), 197
- `load()` (*PrivBayes static method*), 76
- `load()` (*PrivBayesPlugin static method*), 79
- `load()` (*RadialGANPlugin static method*), 108
- `load()` (*RTVAEPlugin static method*), 60
- `load()` (*Serializable static method*), 215
- `load()` (*SurVAEPlugin static method*), 134
- `load()` (*SurvivalCTGANPlugin static method*), 130
- `load()` (*SurvivalFunctionTimeToEvent static method*), 473
- `load()` (*SurvivalGANPlugin static method*), 126
- `load()` (*SurvivalNFlowPlugin static method*), 138
- `load()` (*Teachers static method*), 75
- `load()` (*TimeGANPlugin static method*), 143
- `load()` (*TimeVAEPlugin static method*), 153
- `load()` (*TVAEPlugin static method*), 64
- `load()` (*WeibullAFTSurvivalAnalysis static method*), 441
- `load()` (*XGBSurvivalAnalysis static method*), 443
- `load()` (*XGBTimeSeriesSurvival static method*), 459
- `load_dict()` (*AdsGANPlugin static method*), 69
- `load_dict()` (*ARFPlugin static method*), 27
- `load_dict()` (*BayesianNetworkPlugin static method*), 46
- `load_dict()` (*CoxPHSurvivalAnalysis static method*), 442
- `load_dict()` (*CoxTimeSeriesSurvival static method*), 444
- `load_dict()` (*CTGANPlugin static method*), 51
- `load_dict()` (*DATETimeToEvent static method*), 460
- `load_dict()` (*DECAFPlugin static method*), 89
- `load_dict()` (*DeepHitSurvivalAnalysis static method*), 442
- `load_dict()` (*DPGANPlugin static method*), 84
- `load_dict()` (*DynamicDeepHitTimeSeriesSurvival static method*), 458
- `load_dict()` (*FourierFlowsPlugin static method*), 148
- `load_dict()` (*GOGGLEPlugin static method*), 23
- `load_dict()` (*ImageAdsGANPlugin static method*), 162
- `load_dict()` (*ImageCGANPlugin static method*), 157
- `load_dict()` (*NormalizingFlowsPlugin static method*), 56
- `load_dict()` (*PATEGAN static method*), 70
- `load_dict()` (*PATEGANPlugin static method*), 74
- `load_dict()` (*Plugin static method*), 196
- `load_dict()` (*PrivBayes static method*), 76
- `load_dict()` (*PrivBayesPlugin static method*), 79
- `load_dict()` (*RadialGANPlugin static method*), 108
- `load_dict()` (*RTVAEPlugin static method*), 60
- `load_dict()` (*Serializable static method*), 215
- `load_dict()` (*SurVAEPlugin static method*), 134
- `load_dict()` (*SurvivalCTGANPlugin static method*), 130
- `load_dict()` (*SurvivalFunctionTimeToEvent static method*), 473
- `load_dict()` (*SurvivalGANPlugin static method*), 126
- `load_dict()` (*SurvivalNFlowPlugin static method*), 138
- `load_dict()` (*Teachers static method*), 75
- `load_dict()` (*TimeGANPlugin static method*), 143
- `load_dict()` (*TimeVAEPlugin static method*), 153
- `load_dict()` (*TVAEPlugin static method*), 64
- `load_dict()` (*WeibullAFTSurvivalAnalysis static method*), 441
- `load_dict()` (*XGBSurvivalAnalysis static method*), 443
- `load_dict()` (*XGBTimeSeriesSurvival static method*), 459
- `load_state_dict()` (*ConditionalDiscriminator method*), 479
- `load_state_dict()` (*ConditionalGenerator method*), 492
- `load_state_dict()` (*ConvNet method*), 505
- `load_state_dict()` (*Decoder method*), 280
- `load_state_dict()` (*DynamicDeepHitLayers method*), 448
- `load_state_dict()` (*Encoder method*), 292
- `load_state_dict()` (*GAN method*), 252
- `load_state_dict()` (*ImageGAN method*), 521
- `load_state_dict()` (*LatentODE method*), 391
- `load_state_dict()` (*LinearLayer method*), 224
- `load_state_dict()` (*MLP method*), 237
- `load_state_dict()` (*NormalizingFlows method*), 267
- `load_state_dict()` (*RadialGAN method*), 96
- `load_state_dict()` (*TabularFlows method*), 363
- `load_state_dict()` (*TabularGAN method*), 335
- `load_state_dict()` (*TabularRadialGAN method*), 114
- `load_state_dict()` (*TabularVAE method*), 349
- `load_state_dict()` (*TimeEventGAN method*), 465
- `load_state_dict()` (*TimeSeriesDecoder method*), 404
- `load_state_dict()` (*TimeSeriesEncoder method*), 417
- `load_state_dict()` (*TimeSeriesGAN method*), 378
- `load_state_dict()` (*TimeSeriesVAE method*), 432
- `load_state_dict()` (*TransformerModel method*), 319

- load\_state\_dict() (VAE method), 306
- LogDistribution (class in *synthcity.plugins.core.distribution*), 211
- LogDistribution.Config (class in *synthcity.plugins.core.distribution*), 211
- longitudinal\_loss() (DynamicDeepHitModel method), 457
- low (DatetimeDistribution attribute), 202
- low (FloatDistribution attribute), 206
- low (IntegerDistribution attribute), 210
- low (IntLogDistribution attribute), 208
- low (LogDistribution attribute), 212
- ## M
- map\_nonlin() (in module *synthcity.plugins.core.models.convnet*), 513
- marginal\_distribution (CategoricalDistribution attribute), 200
- marginal\_distribution (DatetimeDistribution attribute), 202
- marginal\_distribution (Distribution attribute), 205
- marginal\_distribution (FloatDistribution attribute), 206
- marginal\_distribution (IntegerDistribution attribute), 210
- marginal\_distribution (IntLogDistribution attribute), 208
- marginal\_distribution (LogDistribution attribute), 212
- marginal\_probabilities() (CategoricalDistribution method), 201
- marginal\_probabilities() (DatetimeDistribution method), 202
- marginal\_probabilities() (Distribution method), 205
- marginal\_probabilities() (FloatDistribution method), 206
- marginal\_probabilities() (IntegerDistribution method), 210
- marginal\_probabilities() (IntLogDistribution method), 208
- marginal\_probabilities() (LogDistribution method), 212
- marginal\_states() (CategoricalDistribution method), 201
- marginal\_states() (DatetimeDistribution method), 202
- marginal\_states() (Distribution method), 205
- marginal\_states() (FloatDistribution method), 206
- marginal\_states() (IntegerDistribution method), 210
- marginal\_states() (IntLogDistribution method), 208
- marginal\_states() (LogDistribution method), 212
- mask\_temporal\_data() (TimeSeriesDataLoader static method), 36
- mask\_temporal\_data() (TimeSeriesSurvivalDataLoader static method), 39
- match() (Constraints method), 199
- match() (DataLoader method), 30
- match() (GenericDataLoader method), 32
- match() (ImageDataLoader method), 33
- match() (SurvivalAnalysisDataLoader method), 35
- match() (TimeSeriesDataLoader method), 36
- match() (TimeSeriesSurvivalDataLoader method), 39
- max() (CategoricalDistribution method), 201
- max() (DatetimeDistribution method), 203
- max() (Distribution method), 205
- max() (FloatDistribution method), 206
- max() (IntegerDistribution method), 210
- max() (IntLogDistribution method), 208
- max() (LogDistribution method), 212
- MaximumMeanDiscrepancy (class in *synthcity.metrics.eval\_statistical*), 171
- metrics() (AlphaPrecision method), 169
- min() (CategoricalDistribution method), 201
- min() (DatetimeDistribution method), 203
- min() (Distribution method), 205
- min() (FloatDistribution method), 207
- min() (IntegerDistribution method), 210
- min() (IntLogDistribution method), 208
- min() (LogDistribution method), 212
- MLP (class in *synthcity.plugins.core.models.mlp*), 232
- module
- synthcity.benchmark*, 189
  - synthcity.benchmark.utils*, 190
  - synthcity.metrics.eval\_attacks*, 187
  - synthcity.metrics.eval\_detection*, 178
  - synthcity.metrics.eval\_performance*, 174
  - synthcity.metrics.eval\_privacy*, 181
  - synthcity.metrics.eval\_sanity*, 165
  - synthcity.metrics.eval\_statistical*, 168
  - synthcity.metrics.weighted\_metrics*, 189
  - synthcity.plugins.core.constraints*, 197
  - synthcity.plugins.core.dataloader*, 29
  - synthcity.plugins.core.dataset*, 40
  - synthcity.plugins.core.distribution*, 199
  - synthcity.plugins.core.models.convnet*, 474
  - synthcity.plugins.core.models.flows*, 261
  - synthcity.plugins.core.models.gan*, 246
  - synthcity.plugins.core.models.image\_gan*, 515
  - synthcity.plugins.core.models.mlp*, 219
  - synthcity.plugins.core.models.survival\_analysis.surv\_a*, 441
  - synthcity.plugins.core.models.survival\_analysis.surv\_c*, 441
  - synthcity.plugins.core.models.survival\_analysis.surv\_d*, 442



synthcity.plugins.core.models.survival\_analysis.plugin\_privbayes, 443  
 synthcity.plugins.core.models.survival\_analysis.plugin\_survae, 215  
 synthcity.plugins.core.models.tabular\_flows, 358  
 synthcity.plugins.core.models.tabular\_gan, 328  
 synthcity.plugins.core.models.tabular\_vae, 343  
 synthcity.plugins.core.models.time\_series\_survival\_analysis.plugin\_fflows, 443  
 synthcity.plugins.core.models.time\_series\_survival\_analysis.plugin\_timegan, 444  
 synthcity.plugins.core.models.time\_series\_survival\_analysis.plugin\_timevae, 459  
 synthcity.plugins.core.models.time\_to\_event\_model, 459  
 synthcity.plugins.core.models.time\_to\_event\_model\_survival, 473  
 synthcity.plugins.core.models.transformer, 315  
 synthcity.plugins.core.models.ts\_gan, 372  
 synthcity.plugins.core.models.ts\_vae, 387  
 synthcity.plugins.core.models.vae, 275  
 synthcity.plugins.core.plugin, 193  
 synthcity.plugins.core.schema, 213  
 synthcity.plugins.core.serializable, 215  
 synthcity.plugins.domain\_adaptation.plugin\_image\_adsgan, 90  
 synthcity.plugins.generic.plugin\_arf, 24  
 synthcity.plugins.generic.plugin\_bayesian\_network, 43  
 synthcity.plugins.generic.plugin\_ctgan, 47  
 synthcity.plugins.generic.plugin\_goggle, 21  
 synthcity.plugins.generic.plugin\_nflow, 52  
 synthcity.plugins.generic.plugin\_rtvae, 57  
 synthcity.plugins.generic.plugin\_tvae, 61  
 synthcity.plugins.images.plugin\_image\_adsgan, 158  
 synthcity.plugins.images.plugin\_image\_cgan, 154  
 synthcity.plugins.privacy.plugin\_adsgan, 65  
 synthcity.plugins.privacy.plugin\_decaf, 85  
 synthcity.plugins.privacy.plugin\_dpGAN, 81  
 synthcity.plugins.privacy.plugin\_pategan, 70

**N**  
 name (ConditionalDiscriminator method), 479  
 modules() (ConditionalGenerator method), 492  
 modules() (ConditionalGenerator method), 505  
 modules() (Decoder method), 280  
 modules() (DynamicDeepHitLayers method), 449  
 modules() (Encoder method), 293  
 modules() (GAN method), 253  
 modules() (ImageGAN method), 521  
 modules() (LatentODE method), 392  
 modules() (LinearLayer method), 224  
 modules() (MLP method), 238  
 modules() (NormalizingFlows method), 267  
 modules() (RadialGAN method), 96  
 modules() (TabularFlows method), 364  
 modules() (TabularGAN method), 335  
 modules() (TabularRadialGAN method), 115  
 modules() (TabularVAE method), 349  
 modules() (TimeEventGAN method), 465  
 modules() (TimeSeriesDecoder method), 405  
 modules() (TimeSeriesEncoder method), 418  
 modules() (TimeSeriesGAN method), 379  
 modules() (TimeSeriesVAE method), 433  
 modules() (TransformerModel method), 320  
 modules() (VAE method), 307  
 mutual\_info\_score() (PrivBayes method), 76  
 n\_features() (BinEncoder method), 216  
 n\_features() (TabularEncoder method), 218  
 n\_features() (TimeSeriesTabularEncoder method), 219  
 name (CategoricalDistribution attribute), 201  
 name (DatetimeDistribution attribute), 203  
 name (Distribution attribute), 205  
 name (FeatureInfo attribute), 217  
 name (FloatDistribution attribute), 207  
 name (IntegerDistribution attribute), 210  
 name (IntLogDistribution attribute), 208

name (*LogDistribution* attribute), 212  
 name() (*AdsGANPlugin* static method), 69  
 name() (*AlphaPrecision* static method), 169  
 name() (*ARFPlugin* static method), 27  
 name() (*AttackEvaluator* static method), 187  
 name() (*AugmentationPerformanceEvaluatorLinear* static method), 174  
 name() (*AugmentationPerformanceEvaluatorMLP* static method), 175  
 name() (*AugmentationPerformanceEvaluatorXGB* static method), 175  
 name() (*BasicMetricEvaluator* static method), 165  
 name() (*BayesianNetworkPlugin* static method), 46  
 name() (*ChiSquaredTest* static method), 169  
 name() (*CloseValuesProbability* static method), 166  
 name() (*CommonRowsProportion* static method), 166  
 name() (*CoxPHSurvivalAnalysis* static method), 442  
 name() (*CoxTimeSeriesSurvival* static method), 444  
 name() (*CTGANPlugin* static method), 51  
 name() (*DataLeakageLinear* static method), 187  
 name() (*DataLeakageMLP* static method), 188  
 name() (*DataLeakageXGB* static method), 188  
 name() (*DataMismatchScore* static method), 167  
 name() (*DATETimeToEvent* static method), 460  
 name() (*DECAFPlugin* static method), 89  
 name() (*DeephitSurvivalAnalysis* static method), 442  
 name() (*DeltaPresence* static method), 182  
 name() (*DetectionEvaluator* static method), 179  
 name() (*DistantValuesProbability* static method), 167  
 name() (*DomiasMIA* static method), 182  
 name() (*DomiasMIABNAF* static method), 183  
 name() (*DomiasMIAKDE* static method), 183  
 name() (*DomiasMIAPrior* static method), 183  
 name() (*DPGANPlugin* static method), 85  
 name() (*DynamicDeephitTimeSeriesSurvival* static method), 458  
 name() (*FeatureImportanceRankDistance* static method), 176  
 name() (*FourierFlowsPlugin* static method), 148  
 name() (*FrechetInceptionDistance* static method), 170  
 name() (*GOGGLEPlugin* static method), 24  
 name() (*IdentifiabilityScore* static method), 184  
 name() (*ImageAdsGANPlugin* static method), 162  
 name() (*ImageCGANPlugin* static method), 157  
 name() (*InverseKLDivergence* static method), 170  
 name() (*JensenShannonDistance* static method), 171  
 name() (*kAnonymization* static method), 185  
 name() (*kMap* static method), 186  
 name() (*KolmogorovSmirnovTest* static method), 171  
 name() (*IDiversityDistinct* static method), 186  
 name() (*MaximumMeanDiscrepancy* static method), 172  
 name() (*NearestSyntheticNeighborDistance* static method), 168  
 name() (*NormalizingFlowsPlugin* static method), 56  
 name() (*PATEGANPlugin* static method), 74  
 name() (*PerformanceEvaluator* static method), 176  
 name() (*PerformanceEvaluatorLinear* static method), 177  
 name() (*PerformanceEvaluatorMLP* static method), 177  
 name() (*PerformanceEvaluatorXGB* static method), 178  
 name() (*Plugin* static method), 196  
 name() (*PRDCScore* static method), 172  
 name() (*PrivacyEvaluator* static method), 185  
 name() (*PrivBayesPlugin* static method), 79  
 name() (*RadialGANPlugin* static method), 108  
 name() (*RTVAEPlugin* static method), 60  
 name() (*StatisticalEvaluator* static method), 173  
 name() (*SurVAEPlugin* static method), 134  
 name() (*SurvivalCTGANPlugin* static method), 130  
 name() (*SurvivalFunctionTimeToEvent* static method), 473  
 name() (*SurvivalGANPlugin* static method), 126  
 name() (*SurvivalKMDistance* static method), 173  
 name() (*SurvivalNFlowPlugin* static method), 138  
 name() (*SyntheticDetectionGMM* static method), 179  
 name() (*SyntheticDetectionLinear* static method), 180  
 name() (*SyntheticDetectionMLP* static method), 180  
 name() (*SyntheticDetectionXGB* static method), 181  
 name() (*TimeGANPlugin* static method), 143  
 name() (*TimeVAEPlugin* static method), 153  
 name() (*TVAEPlugin* static method), 64  
 name() (*WassersteinDistance* static method), 174  
 name() (*WeibullAFTSurvivalAnalysis* static method), 441  
 name() (*XGBSurvivalAnalysis* static method), 443  
 name() (*XGBTimeSeriesSurvival* static method), 459  
 named\_buffers() (*ConditionalDiscriminator* method), 479  
 named\_buffers() (*ConditionalGenerator* method), 492  
 named\_buffers() (*ConvNet* method), 506  
 named\_buffers() (*Decoder* method), 280  
 named\_buffers() (*DynamicDeepHitLayers* method), 449  
 named\_buffers() (*Encoder* method), 293  
 named\_buffers() (*GAN* method), 253  
 named\_buffers() (*ImageGAN* method), 522  
 named\_buffers() (*LatentODE* method), 392  
 named\_buffers() (*LinearLayer* method), 224  
 named\_buffers() (*MLP* method), 238  
 named\_buffers() (*NormalizingFlows* method), 267  
 named\_buffers() (*RadialGAN* method), 97  
 named\_buffers() (*TabularFlows* method), 364  
 named\_buffers() (*TabularGAN* method), 336  
 named\_buffers() (*TabularRadialGAN* method), 115  
 named\_buffers() (*TabularVAE* method), 350  
 named\_buffers() (*TimeEventGAN* method), 466  
 named\_buffers() (*TimeSeriesDecoder* method), 405  
 named\_buffers() (*TimeSeriesEncoder* method), 418  
 named\_buffers() (*TimeSeriesGAN* method), 379

- `named_buffers()` (*TimeSeriesVAE method*), 433  
`named_buffers()` (*TransformerModel method*), 320  
`named_buffers()` (*VAE method*), 307  
`named_children()` (*ConditionalDiscriminator method*), 480  
`named_children()` (*ConditionalGenerator method*), 493  
`named_children()` (*ConvNet method*), 506  
`named_children()` (*Decoder method*), 281  
`named_children()` (*DynamicDeepHitLayers method*), 450  
`named_children()` (*Encoder method*), 293  
`named_children()` (*GAN method*), 254  
`named_children()` (*ImageGAN method*), 522  
`named_children()` (*LatentODE method*), 393  
`named_children()` (*LinearLayer method*), 225  
`named_children()` (*MLP method*), 238  
`named_children()` (*NormalizingFlows method*), 268  
`named_children()` (*RadialGAN method*), 97  
`named_children()` (*TabularFlows method*), 364  
`named_children()` (*TabularGAN method*), 336  
`named_children()` (*TabularRadialGAN method*), 115  
`named_children()` (*TabularVAE method*), 350  
`named_children()` (*TimeEventGAN method*), 466  
`named_children()` (*TimeSeriesDecoder method*), 406  
`named_children()` (*TimeSeriesEncoder method*), 418  
`named_children()` (*TimeSeriesGAN method*), 380  
`named_children()` (*TimeSeriesVAE method*), 433  
`named_children()` (*TransformerModel method*), 321  
`named_children()` (*VAE method*), 308  
`named_modules()` (*ConditionalDiscriminator method*), 480  
`named_modules()` (*ConditionalGenerator method*), 493  
`named_modules()` (*ConvNet method*), 506  
`named_modules()` (*Decoder method*), 281  
`named_modules()` (*DynamicDeepHitLayers method*), 450  
`named_modules()` (*Encoder method*), 294  
`named_modules()` (*GAN method*), 254  
`named_modules()` (*ImageGAN method*), 522  
`named_modules()` (*LatentODE method*), 393  
`named_modules()` (*LinearLayer method*), 225  
`named_modules()` (*MLP method*), 239  
`named_modules()` (*NormalizingFlows method*), 268  
`named_modules()` (*RadialGAN method*), 97  
`named_modules()` (*TabularFlows method*), 365  
`named_modules()` (*TabularGAN method*), 336  
`named_modules()` (*TabularRadialGAN method*), 115  
`named_modules()` (*TabularVAE method*), 350  
`named_modules()` (*TimeEventGAN method*), 466  
`named_modules()` (*TimeSeriesDecoder method*), 406  
`named_modules()` (*TimeSeriesEncoder method*), 419  
`named_modules()` (*TimeSeriesGAN method*), 380  
`named_modules()` (*TimeSeriesVAE method*), 434  
`named_modules()` (*TransformerModel method*), 321  
`named_modules()` (*VAE method*), 308  
`named_parameters()` (*ConditionalDiscriminator method*), 480  
`named_parameters()` (*ConditionalGenerator method*), 493  
`named_parameters()` (*ConvNet method*), 507  
`named_parameters()` (*Decoder method*), 281  
`named_parameters()` (*DynamicDeepHitLayers method*), 450  
`named_parameters()` (*Encoder method*), 294  
`named_parameters()` (*GAN method*), 254  
`named_parameters()` (*ImageGAN method*), 523  
`named_parameters()` (*LatentODE method*), 393  
`named_parameters()` (*LinearLayer method*), 225  
`named_parameters()` (*MLP method*), 239  
`named_parameters()` (*NormalizingFlows method*), 268  
`named_parameters()` (*RadialGAN method*), 98  
`named_parameters()` (*TabularFlows method*), 365  
`named_parameters()` (*TabularGAN method*), 337  
`named_parameters()` (*TabularRadialGAN method*), 116  
`named_parameters()` (*TabularVAE method*), 351  
`named_parameters()` (*TimeEventGAN method*), 467  
`named_parameters()` (*TimeSeriesDecoder method*), 406  
`named_parameters()` (*TimeSeriesEncoder method*), 419  
`named_parameters()` (*TimeSeriesGAN method*), 380  
`named_parameters()` (*TimeSeriesVAE method*), 434  
`named_parameters()` (*TransformerModel method*), 321  
`named_parameters()` (*VAE method*), 308  
`NearestSyntheticNeighborDistance` (*class in synthcity.metrics.eval\_sanity*), 167  
`negative_log_likelihood()` (*DynamicDeepHit-Model method*), 457  
`network_edge` (*class in synthcity.plugins.privacy.plugin\_privbayer*), 80  
`NormalizingFlows` (*class in synthcity.plugins.core.models.flows*), 261  
`NormalizingFlowsPlugin` (*class in synthcity.plugins.generic.plugin\_nflow*), 52  
`NormalizingFlowsPlugin.Config` (*class in synthcity.plugins.generic.plugin\_nflow*), 54  
`numpy()` (*DataLoader method*), 30  
`numpy()` (*FlexibleDataset method*), 40  
`numpy()` (*GenericDataLoader method*), 32  
`numpy()` (*ImageDataLoader method*), 33  
`numpy()` (*SurvivalAnalysisDataLoader method*), 35  
`numpy()` (*TimeSeriesDataLoader method*), 36  
`numpy()` (*TimeSeriesSurvivalDataLoader method*), 39  
`NumpyDataset` (*class in synthcity.plugins.core.dataset*), 40



## O

offset (*DatetimeDistribution* attribute), 203  
 output\_dimensions (*FeatureInfo* attribute), 217

## P

pack\_raw\_data() (*TimeSeriesDataLoader* static method), 37  
 pack\_raw\_data() (*TimeSeriesSurvivalDataLoader* static method), 39  
 pad\_and\_mask() (*TimeSeriesDataLoader* static method), 37  
 pad\_and\_mask() (*TimeSeriesSurvivalDataLoader* static method), 39  
 pad\_raw\_data() (*TimeSeriesDataLoader* static method), 37  
 pad\_raw\_data() (*TimeSeriesSurvivalDataLoader* static method), 39  
 pad\_raw\_features() (*TimeSeriesDataLoader* static method), 37  
 pad\_raw\_features() (*TimeSeriesSurvivalDataLoader* static method), 39  
 parameters() (*ConditionalDiscriminator* method), 481  
 parameters() (*ConditionalGenerator* method), 494  
 parameters() (*ConvNet* method), 507  
 parameters() (*Decoder* method), 282  
 parameters() (*DynamicDeepHitLayers* method), 451  
 parameters() (*Encoder* method), 294  
 parameters() (*GAN* method), 255  
 parameters() (*ImageGAN* method), 523  
 parameters() (*LatentODE* method), 394  
 parameters() (*LinearLayer* method), 226  
 parameters() (*MLP* method), 239  
 parameters() (*NormalizingFlows* method), 269  
 parameters() (*RadialGAN* method), 98  
 parameters() (*TabularFlows* method), 366  
 parameters() (*TabularGAN* method), 337  
 parameters() (*TabularRadialGAN* method), 116  
 parameters() (*TabularVAE* method), 351  
 parameters() (*TimeEventGAN* method), 467  
 parameters() (*TimeSeriesDecoder* method), 407  
 parameters() (*TimeSeriesEncoder* method), 420  
 parameters() (*TimeSeriesGAN* method), 381  
 parameters() (*TimeSeriesVAE* method), 435  
 parameters() (*TransformerModel* method), 322  
 parameters() (*VAE* method), 309  
 parents (*network\_edge* attribute), 80  
 parse\_file() (*CategoricalDistribution* class method), 201  
 parse\_file() (*Constraints* class method), 199  
 parse\_file() (*DatetimeDistribution* class method), 203  
 parse\_file() (*Distribution* class method), 205  
 parse\_file() (*FeatureInfo* class method), 217  
 parse\_file() (*FloatDistribution* class method), 207  
 parse\_file() (*IntegerDistribution* class method), 210

parse\_file() (*IntLogDistribution* class method), 208  
 parse\_file() (*LogDistribution* class method), 212  
 parse\_file() (*Schema* class method), 214  
 parse\_obj() (*CategoricalDistribution* class method), 201  
 parse\_obj() (*Constraints* class method), 199  
 parse\_obj() (*DatetimeDistribution* class method), 203  
 parse\_obj() (*Distribution* class method), 205  
 parse\_obj() (*FeatureInfo* class method), 217  
 parse\_obj() (*FloatDistribution* class method), 207  
 parse\_obj() (*IntegerDistribution* class method), 210  
 parse\_obj() (*IntLogDistribution* class method), 208  
 parse\_obj() (*LogDistribution* class method), 212  
 parse\_obj() (*Schema* class method), 214  
 parse\_raw() (*CategoricalDistribution* class method), 201  
 parse\_raw() (*Constraints* class method), 199  
 parse\_raw() (*DatetimeDistribution* class method), 203  
 parse\_raw() (*Distribution* class method), 205  
 parse\_raw() (*FeatureInfo* class method), 217  
 parse\_raw() (*FloatDistribution* class method), 207  
 parse\_raw() (*IntegerDistribution* class method), 210  
 parse\_raw() (*IntLogDistribution* class method), 208  
 parse\_raw() (*LogDistribution* class method), 212  
 parse\_raw() (*Schema* class method), 214  
 pate\_lamda() (*Teachers* method), 75  
 PATEGAN (class in *synthcity.plugins.privacy.plugin\_pategan*), 70  
 PATEGANPlugin (class in *synthcity.plugins.privacy.plugin\_pategan*), 71  
 PATEGANPlugin.Config (class in *synthcity.plugins.privacy.plugin\_pategan*), 72  
 PerformanceEvaluator (class in *synthcity.metrics.eval\_performance*), 176  
 PerformanceEvaluatorLinear (class in *synthcity.metrics.eval\_performance*), 176  
 PerformanceEvaluatorMLP (class in *synthcity.metrics.eval\_performance*), 177  
 PerformanceEvaluatorXGB (class in *synthcity.metrics.eval\_performance*), 178  
 plot() (*AdsGANPlugin* method), 69  
 plot() (*ARFPlugin* method), 27  
 plot() (*BayesianNetworkPlugin* method), 46  
 plot() (*CTGANPlugin* method), 51  
 plot() (*DECAFPlugin* method), 90  
 plot() (*DPGANPlugin* method), 85  
 plot() (*FourierFlowsPlugin* method), 148  
 plot() (*GOGGLEPlugin* method), 24  
 plot() (*ImageAdsGANPlugin* method), 162  
 plot() (*ImageCGANPlugin* method), 157  
 plot() (*NormalizingFlowsPlugin* method), 56  
 plot() (*PATEGANPlugin* method), 74  
 plot() (*Plugin* method), 196  
 plot() (*PrivBayesPlugin* method), 79

- `plot()` (*RadialGANPlugin* method), 108  
`plot()` (*RTVAEPlugin* method), 60  
`plot()` (*SurVAEPlugin* method), 134  
`plot()` (*SurvivalCTGANPlugin* method), 130  
`plot()` (*SurvivalGANPlugin* method), 126  
`plot()` (*SurvivalNFlowPlugin* method), 138  
`plot()` (*TimeGANPlugin* method), 143  
`plot()` (*TimeVAEPlugin* method), 153  
`plot()` (*TVAEPlugin* method), 64  
`Plugin` (class in *synthcity.plugins.core.plugin*), 193  
`plugin` (in module *synthcity.plugins.domain\_adaptation.plugin\_radialgan*), 122  
`plugin` (in module *synthcity.plugins.generic.plugin\_arf*), 28  
`plugin` (in module *synthcity.plugins.generic.plugin\_bayesian\_network*), 47  
`plugin` (in module *synthcity.plugins.generic.plugin\_ctgan*), 52  
`plugin` (in module *synthcity.plugins.generic.plugin\_goggle*), 24  
`plugin` (in module *synthcity.plugins.generic.plugin\_nflow*), 57  
`plugin` (in module *synthcity.plugins.generic.plugin\_rtvae*), 61  
`plugin` (in module *synthcity.plugins.generic.plugin\_tvae*), 65  
`plugin` (in module *synthcity.plugins.images.plugin\_image\_adsgan*), 163  
`plugin` (in module *synthcity.plugins.images.plugin\_image\_cgan*), 158  
`plugin` (in module *synthcity.plugins.privacy.plugin\_adsgan*), 70  
`plugin` (in module *synthcity.plugins.privacy.plugin\_decaf*), 90  
`plugin` (in module *synthcity.plugins.privacy.plugin\_dpGAN*), 85  
`plugin` (in module *synthcity.plugins.privacy.plugin\_pategan*), 75  
`plugin` (in module *synthcity.plugins.privacy.plugin\_privbayes*), 80  
`plugin` (in module *synthcity.plugins.survival\_analysis.plugin\_survae*), 135  
`plugin` (in module *synthcity.plugins.survival\_analysis.plugin\_survival\_ctgan*), 131  
`plugin` (in module *synthcity.plugins.survival\_analysis.plugin\_survival\_gan*), 127  
`plugin` (in module *synthcity.plugins.survival\_analysis.plugin\_survival\_nflow*), 139  
`plugin` (in module *synthcity.plugins.time\_series.plugin\_fflows*), 148  
`plugin` (in module *synthcity.plugins.time\_series.plugin\_timegan*), 144  
`plugin` (in module *synthcity.plugins.time\_series.plugin\_timevae*), 153  
`Plugin.Config` (class in *synthcity.plugins.core.plugin*), 194  
`PluginLoader` (class in *synthcity.plugins.core.plugin*), 196  
`PRDCScore` (class in *synthcity.metrics.eval\_statistical*), 172  
`predict()` (*ConvNet* method), 507  
`predict()` (*CoxPHSurvivalAnalysis* method), 442  
`predict()` (*CoxTimeSeriesSurvival* method), 444  
`predict()` (*DATETimeToEvent* method), 460  
`predict()` (*DeepHitSurvivalAnalysis* method), 442  
`predict()` (*DynamicDeepHitTimeSeriesSurvival* method), 458  
`predict()` (*MLP* method), 240  
`predict()` (*SurvivalFunctionTimeToEvent* method), 474  
`predict()` (*WeibullAFTSurvivalAnalysis* method), 441  
`predict()` (*XGBSurvivalAnalysis* method), 443  
`predict()` (*XGBTimeSeriesSurvival* method), 459  
`predict_any()` (*DATETimeToEvent* method), 460  
`predict_any()` (*SurvivalFunctionTimeToEvent* method), 474  
`predict_emb()` (*DynamicDeepHitModel* method), 457  
`predict_emb()` (*DynamicDeepHitTimeSeriesSurvival* method), 458  
`predict_proba()` (*ConvNet* method), 507  
`predict_proba()` (*MLP* method), 240  
`predict_risk()` (*DynamicDeepHitModel* method), 457  
`predict_survival()` (*DynamicDeepHitModel* method), 457  
`print()` (*Benchmarks* static method), 190  
`print_score()` (in module *synthcity.benchmark*), 190  
`PrivacyEvaluator` (class in *synthcity.metrics.eval\_privacy*), 184  
`PrivBayes` (class in *synthcity.plugins.privacy.plugin\_privbayes*), 76  
`PrivBayesPlugin` (class in *synthcity.plugins.privacy.plugin\_privbayes*), 76  
`PrivBayesPlugin.Config` (class in *synthcity.plugins.privacy.plugin\_privbayes*), 77  
`protected_cols` (Schema attribute), 214
- ## R
- `RadialGAN` (class in *synthcity.plugins.domain\_adaptation.plugin\_radialgan*), 90

- RadialGANPlugin (class in synthcity.plugins.domain\_adaptation.plugin\_radialgan), 104
- RadialGANPlugin.Config (class in synthcity.plugins.domain\_adaptation.plugin\_radialgan), 106
- random\_state (CategoricalDistribution attribute), 201
- random\_state (DatetimeDistribution attribute), 203
- random\_state (Distribution attribute), 205
- random\_state (FloatDistribution attribute), 207
- random\_state (IntegerDistribution attribute), 210
- random\_state (IntLogDistribution attribute), 208
- random\_state (LogDistribution attribute), 212
- random\_state (Schema attribute), 214
- ranking\_loss() (DynamicDeepHitModel method), 458
- raw() (DataLoader method), 30
- raw() (GenericDataLoader method), 32
- raw() (ImageDataLoader method), 33
- raw() (SurvivalAnalysisDataLoader method), 35
- raw() (TimeSeriesDataLoader method), 37
- raw() (TimeSeriesSurvivalDataLoader method), 39
- raw\_columns (TimeSeriesDataLoader property), 37
- raw\_columns (TimeSeriesSurvivalDataLoader property), 39
- reduction() (AlphaPrecision method), 169
- reduction() (AttackEvaluator method), 187
- reduction() (AugmentationPerformanceEvaluatorLinear method), 174
- reduction() (AugmentationPerformanceEvaluatorMLP method), 175
- reduction() (AugmentationPerformanceEvaluatorXGB method), 175
- reduction() (BasicMetricEvaluator method), 165
- reduction() (ChiSquaredTest method), 169
- reduction() (CloseValuesProbability method), 166
- reduction() (CommonRowsProportion method), 166
- reduction() (DataLeakageLinear method), 188
- reduction() (DataLeakageMLP method), 188
- reduction() (DataLeakageXGB method), 189
- reduction() (DataMismatchScore method), 167
- reduction() (DeltaPresence method), 182
- reduction() (DetectionEvaluator method), 179
- reduction() (DistantValuesProbability method), 167
- reduction() (DomiasMIA method), 182
- reduction() (DomiasMIABNAF method), 183
- reduction() (DomiasMIAKDE method), 183
- reduction() (DomiasMIAPrior method), 184
- reduction() (FeatureImportanceRankDistance method), 176
- reduction() (FrechetInceptionDistance method), 170
- reduction() (IdentifiabilityScore method), 184
- reduction() (InverseKLDivergence method), 170
- reduction() (JensenShannonDistance method), 171
- reduction() (kAnonymization method), 185
- reduction() (kMap method), 186
- reduction() (KolmogorovSmirnovTest method), 171
- reduction() (LDiversityDistinct method), 186
- reduction() (MaximumMeanDiscrepancy method), 172
- reduction() (NearestSyntheticNeighborDistance method), 168
- reduction() (PerformanceEvaluator method), 176
- reduction() (PerformanceEvaluatorLinear method), 177
- reduction() (PerformanceEvaluatorMLP method), 177
- reduction() (PerformanceEvaluatorXGB method), 178
- reduction() (PRDCScore method), 172
- reduction() (PrivacyEvaluator method), 185
- reduction() (StatisticalEvaluator method), 173
- reduction() (SurvivalKMDistance method), 173
- reduction() (SyntheticDetectionGMM method), 179
- reduction() (SyntheticDetectionLinear method), 180
- reduction() (SyntheticDetectionMLP method), 180
- reduction() (SyntheticDetectionXGB method), 181
- reduction() (WassersteinDistance method), 174
- register\_backward\_hook() (ConditionalDiscriminator method), 481
- register\_backward\_hook() (ConditionalGenerator method), 494
- register\_backward\_hook() (ConvNet method), 507
- register\_backward\_hook() (Decoder method), 282
- register\_backward\_hook() (DynamicDeepHitLayers method), 451
- register\_backward\_hook() (Encoder method), 295
- register\_backward\_hook() (GAN method), 255
- register\_backward\_hook() (ImageGAN method), 523
- register\_backward\_hook() (LatentODE method), 394
- register\_backward\_hook() (LinearLayer method), 226
- register\_backward\_hook() (MLP method), 240
- register\_backward\_hook() (NormalizingFlows method), 269
- register\_backward\_hook() (RadialGAN method), 98
- register\_backward\_hook() (TabularFlows method), 366
- register\_backward\_hook() (TabularGAN method), 337
- register\_backward\_hook() (TabularRadialGAN method), 117
- register\_backward\_hook() (TabularVAE method), 351
- register\_backward\_hook() (TimeEventGAN method), 467
- register\_backward\_hook() (TimeSeriesDecoder method), 407
- register\_backward\_hook() (TimeSeriesEncoder method), 420
- register\_backward\_hook() (TimeSeriesGAN method), 420

- method), 381
- register\_backward\_hook() (TimeSeriesVAE method), 435
- register\_backward\_hook() (TransformerModel method), 322
- register\_backward\_hook() (VAE method), 309
- register\_buffer() (ConditionalDiscriminator method), 481
- register\_buffer() (ConditionalGenerator method), 494
- register\_buffer() (ConvNet method), 508
- register\_buffer() (Decoder method), 282
- register\_buffer() (DynamicDeepHitLayers method), 451
- register\_buffer() (Encoder method), 295
- register\_buffer() (GAN method), 255
- register\_buffer() (ImageGAN method), 524
- register\_buffer() (LatentODE method), 394
- register\_buffer() (LinearLayer method), 226
- register\_buffer() (MLP method), 240
- register\_buffer() (NormalizingFlows method), 269
- register\_buffer() (RadialGAN method), 99
- register\_buffer() (TabularFlows method), 366
- register\_buffer() (TabularGAN method), 338
- register\_buffer() (TabularRadialGAN method), 117
- register\_buffer() (TabularVAE method), 352
- register\_buffer() (TimeEventGAN method), 467
- register\_buffer() (TimeSeriesDecoder method), 407
- register\_buffer() (TimeSeriesEncoder method), 420
- register\_buffer() (TimeSeriesGAN method), 381
- register\_buffer() (TimeSeriesVAE method), 435
- register\_buffer() (TransformerModel method), 322
- register\_buffer() (VAE method), 309
- register\_forward\_hook() (ConditionalDiscriminator method), 482
- register\_forward\_hook() (ConditionalGenerator method), 495
- register\_forward\_hook() (ConvNet method), 508
- register\_forward\_hook() (Decoder method), 283
- register\_forward\_hook() (DynamicDeepHitLayers method), 452
- register\_forward\_hook() (Encoder method), 295
- register\_forward\_hook() (GAN method), 256
- register\_forward\_hook() (ImageGAN method), 524
- register\_forward\_hook() (LatentODE method), 395
- register\_forward\_hook() (LinearLayer method), 227
- register\_forward\_hook() (MLP method), 241
- register\_forward\_hook() (NormalizingFlows method), 270
- register\_forward\_hook() (RadialGAN method), 99
- register\_forward\_hook() (TabularFlows method), 367
- register\_forward\_hook() (TabularGAN method), 338
- register\_forward\_hook() (TabularRadialGAN method), 118
- register\_forward\_hook() (TabularVAE method), 352
- register\_forward\_hook() (TimeEventGAN method), 468
- register\_forward\_hook() (TimeSeriesDecoder method), 408
- register\_forward\_hook() (TimeSeriesEncoder method), 421
- register\_forward\_hook() (TabularRadialGAN method), 117
- register\_forward\_hook() (TabularVAE method), 352
- register\_forward\_hook() (TimeEventGAN method), 468
- register\_forward\_hook() (TimeSeriesDecoder method), 408
- register\_forward\_hook() (TimeSeriesEncoder method), 420
- register\_forward\_hook() (TimeSeriesGAN method), 382
- register\_forward\_hook() (TimeSeriesVAE method), 435
- register\_forward\_hook() (TransformerModel method), 323
- register\_forward\_hook() (VAE method), 310
- register\_forward\_pre\_hook() (ConditionalDiscriminator method), 482
- register\_forward\_pre\_hook() (ConditionalGenerator method), 495
- register\_forward\_pre\_hook() (ConvNet method), 508
- register\_forward\_pre\_hook() (Decoder method), 283
- register\_forward\_pre\_hook() (DynamicDeepHitLayers method), 452
- register\_forward\_pre\_hook() (Encoder method), 296
- register\_forward\_pre\_hook() (GAN method), 256
- register\_forward\_pre\_hook() (ImageGAN method), 524
- register\_forward\_pre\_hook() (LatentODE method), 395
- register\_forward\_pre\_hook() (LinearLayer method), 227
- register\_forward\_pre\_hook() (MLP method), 241
- register\_forward\_pre\_hook() (NormalizingFlows method), 270
- register\_forward\_pre\_hook() (RadialGAN method), 99
- register\_forward\_pre\_hook() (TabularFlows method), 367
- register\_forward\_pre\_hook() (TabularGAN method), 338
- register\_forward\_pre\_hook() (TabularRadialGAN method), 118
- register\_forward\_pre\_hook() (TabularVAE method), 352
- register\_forward\_pre\_hook() (TimeEventGAN method), 468
- register\_forward\_pre\_hook() (TimeSeriesDecoder method), 408
- register\_forward\_pre\_hook() (TimeSeriesEncoder method), 421



<code>register_forward_pre_hook()</code>	( <i>TimeSeriesGAN method</i> ), 382	<code>register_load_state_dict_post_hook()</code>	( <i>ConditionalDiscriminator method</i> ), 483
<code>register_forward_pre_hook()</code>	( <i>TimeSeriesVAE method</i> ), 436	<code>register_load_state_dict_post_hook()</code>	( <i>ConditionalGenerator method</i> ), 496
<code>register_forward_pre_hook()</code>	( <i>TransformerModel method</i> ), 323	<code>register_load_state_dict_post_hook()</code>	( <i>ConvNet method</i> ), 509
<code>register_forward_pre_hook()</code>	( <i>VAE method</i> ), 310	<code>register_load_state_dict_post_hook()</code>	( <i>Decoder method</i> ), 284
<code>register_full_backward_hook()</code>	( <i>ConditionalDiscriminator method</i> ), 482	<code>register_load_state_dict_post_hook()</code>	( <i>DynamicDeepHitLayers method</i> ), 453
<code>register_full_backward_hook()</code>	( <i>ConditionalGenerator method</i> ), 495	<code>register_load_state_dict_post_hook()</code>	( <i>Encoder method</i> ), 296
<code>register_full_backward_hook()</code>	( <i>ConvNet method</i> ), 508	<code>register_load_state_dict_post_hook()</code>	( <i>GAN method</i> ), 257
<code>register_full_backward_hook()</code>	( <i>Decoder method</i> ), 283	<code>register_load_state_dict_post_hook()</code>	( <i>ImageGAN method</i> ), 525
<code>register_full_backward_hook()</code>	( <i>DynamicDeepHitLayers method</i> ), 452	<code>register_load_state_dict_post_hook()</code>	( <i>LatentODE method</i> ), 396
<code>register_full_backward_hook()</code>	( <i>Encoder method</i> ), 296	<code>register_load_state_dict_post_hook()</code>	( <i>LinearLayer method</i> ), 228
<code>register_full_backward_hook()</code>	( <i>GAN method</i> ), 256	<code>register_load_state_dict_post_hook()</code>	( <i>MLP method</i> ), 241
<code>register_full_backward_hook()</code>	( <i>ImageGAN method</i> ), 525	<code>register_load_state_dict_post_hook()</code>	( <i>NormalizingFlows method</i> ), 271
<code>register_full_backward_hook()</code>	( <i>LatentODE method</i> ), 395	<code>register_load_state_dict_post_hook()</code>	( <i>RadialGAN method</i> ), 100
<code>register_full_backward_hook()</code>	( <i>LinearLayer method</i> ), 227	<code>register_load_state_dict_post_hook()</code>	( <i>TabularFlows method</i> ), 368
<code>register_full_backward_hook()</code>	( <i>MLP method</i> ), 241	<code>register_load_state_dict_post_hook()</code>	( <i>TabularGAN method</i> ), 339
<code>register_full_backward_hook()</code>	( <i>NormalizingFlows method</i> ), 270	<code>register_load_state_dict_post_hook()</code>	( <i>TabularRadialGAN method</i> ), 118
<code>register_full_backward_hook()</code>	( <i>RadialGAN method</i> ), 100	<code>register_load_state_dict_post_hook()</code>	( <i>TabularVAE method</i> ), 353
<code>register_full_backward_hook()</code>	( <i>TabularFlows method</i> ), 367	<code>register_load_state_dict_post_hook()</code>	( <i>TimeEventGAN method</i> ), 469
<code>register_full_backward_hook()</code>	( <i>TabularGAN method</i> ), 338	<code>register_load_state_dict_post_hook()</code>	( <i>TimeSeriesDecoder method</i> ), 409
<code>register_full_backward_hook()</code>	( <i>TabularRadialGAN method</i> ), 118	<code>register_load_state_dict_post_hook()</code>	( <i>TimeSeriesEncoder method</i> ), 422
<code>register_full_backward_hook()</code>	( <i>TabularVAE method</i> ), 353	<code>register_load_state_dict_post_hook()</code>	( <i>TimeSeriesGAN method</i> ), 383
<code>register_full_backward_hook()</code>	( <i>TimeEventGAN method</i> ), 468	<code>register_load_state_dict_post_hook()</code>	( <i>TimeSeriesVAE method</i> ), 437
<code>register_full_backward_hook()</code>	( <i>TimeSeriesDecoder method</i> ), 408	<code>register_load_state_dict_post_hook()</code>	( <i>TransformerModel method</i> ), 324
<code>register_full_backward_hook()</code>	( <i>TimeSeriesEncoder method</i> ), 421	<code>register_load_state_dict_post_hook()</code>	( <i>VAE method</i> ), 311
<code>register_full_backward_hook()</code>	( <i>TimeSeriesGAN method</i> ), 382	<code>register_module()</code>	( <i>ConditionalDiscriminator method</i> ), 483
<code>register_full_backward_hook()</code>	( <i>TimeSeriesVAE method</i> ), 436	<code>register_module()</code>	( <i>ConditionalGenerator method</i> ), 496
<code>register_full_backward_hook()</code>	( <i>TransformerModel method</i> ), 323	<code>register_module()</code>	( <i>ConvNet method</i> ), 509
<code>register_full_backward_hook()</code>	( <i>VAE method</i> ), 310	<code>register_module()</code>	( <i>Decoder method</i> ), 284



- `register_module()` (*DynamicDeepHitLayers* method), 453  
`register_module()` (*Encoder* method), 297  
`register_module()` (*GAN* method), 257  
`register_module()` (*ImageGAN* method), 525  
`register_module()` (*LatentODE* method), 396  
`register_module()` (*LinearLayer* method), 228  
`register_module()` (*MLP* method), 242  
`register_module()` (*NormalizingFlows* method), 271  
`register_module()` (*RadialGAN* method), 100  
`register_module()` (*TabularFlows* method), 368  
`register_module()` (*TabularGAN* method), 339  
`register_module()` (*TabularRadialGAN* method), 119  
`register_module()` (*TabularVAE* method), 353  
`register_module()` (*TimeEventGAN* method), 469  
`register_module()` (*TimeSeriesDecoder* method), 409  
`register_module()` (*TimeSeriesEncoder* method), 422  
`register_module()` (*TimeSeriesGAN* method), 383  
`register_module()` (*TimeSeriesVAE* method), 437  
`register_module()` (*TransformerModel* method), 324  
`register_module()` (*VAE* method), 311  
`register_parameter()` (*ConditionalDiscriminator* method), 483  
`register_parameter()` (*ConditionalGenerator* method), 496  
`register_parameter()` (*ConvNet* method), 509  
`register_parameter()` (*Decoder* method), 284  
`register_parameter()` (*DynamicDeepHitLayers* method), 453  
`register_parameter()` (*Encoder* method), 297  
`register_parameter()` (*GAN* method), 257  
`register_parameter()` (*ImageGAN* method), 526  
`register_parameter()` (*LatentODE* method), 396  
`register_parameter()` (*LinearLayer* method), 228  
`register_parameter()` (*MLP* method), 242  
`register_parameter()` (*NormalizingFlows* method), 271  
`register_parameter()` (*RadialGAN* method), 101  
`register_parameter()` (*TabularFlows* method), 368  
`register_parameter()` (*TabularGAN* method), 339  
`register_parameter()` (*TabularRadialGAN* method), 119  
`register_parameter()` (*TabularVAE* method), 354  
`register_parameter()` (*TimeEventGAN* method), 469  
`register_parameter()` (*TimeSeriesDecoder* method), 409  
`register_parameter()` (*TimeSeriesEncoder* method), 422  
`register_parameter()` (*TimeSeriesGAN* method), 383  
`register_parameter()` (*TimeSeriesVAE* method), 437  
`register_parameter()` (*TransformerModel* method), 324  
`register_parameter()` (*VAE* method), 311  
`reload()` (*PluginLoader* method), 197  
`requires_grad_()` (*ConditionalDiscriminator* method), 483  
`requires_grad_()` (*ConditionalGenerator* method), 496  
`requires_grad_()` (*ConvNet* method), 510  
`requires_grad_()` (*Decoder* method), 284  
`requires_grad_()` (*DynamicDeepHitLayers* method), 453  
`requires_grad_()` (*Encoder* method), 297  
`requires_grad_()` (*GAN* method), 257  
`requires_grad_()` (*ImageGAN* method), 526  
`requires_grad_()` (*LatentODE* method), 396  
`requires_grad_()` (*LinearLayer* method), 228  
`requires_grad_()` (*MLP* method), 242  
`requires_grad_()` (*NormalizingFlows* method), 271  
`requires_grad_()` (*RadialGAN* method), 101  
`requires_grad_()` (*TabularFlows* method), 368  
`requires_grad_()` (*TabularGAN* method), 340  
`requires_grad_()` (*TabularRadialGAN* method), 119  
`requires_grad_()` (*TabularVAE* method), 354  
`requires_grad_()` (*TimeEventGAN* method), 470  
`requires_grad_()` (*TimeSeriesDecoder* method), 409  
`requires_grad_()` (*TimeSeriesEncoder* method), 422  
`requires_grad_()` (*TimeSeriesGAN* method), 383  
`requires_grad_()` (*TimeSeriesVAE* method), 437  
`requires_grad_()` (*TransformerModel* method), 324  
`requires_grad_()` (*VAE* method), 311  
`RTVAEPlugin` (class in *synthcity.plugins.generic.plugin\_rtvae*), 57  
`RTVAEPlugin.Config` (class in *synthcity.plugins.generic.plugin\_rtvae*), 58  
`rules` (*Constraints* attribute), 199
- ## S
- `sample()` (*CategoricalDistribution* method), 201  
`sample()` (*DataLoader* method), 30  
`sample()` (*DatetimeDistribution* method), 203  
`sample()` (*Distribution* method), 205  
`sample()` (*FloatDistribution* method), 207  
`sample()` (*GenericDataLoader* method), 32  
`sample()` (*ImageDataLoader* method), 33  
`sample()` (*IntegerDistribution* method), 210  
`sample()` (*IntLogDistribution* method), 208  
`sample()` (*LogDistribution* method), 212  
`sample()` (*PATEGAN* method), 71  
`sample()` (*PrivBayes* method), 76  
`sample()` (*Schema* method), 214  
`sample()` (*SurvivalAnalysisDataLoader* method), 35  
`sample()` (*TimeSeriesDataLoader* method), 37  
`sample()` (*TimeSeriesSurvivalDataLoader* method), 39  
`sample_hyperparameters()` (*AdsGANPlugin* class method), 70  
`sample_hyperparameters()` (*ARFPlugin* class method), 27

<code>sample_hyperparameters()</code> ( <i>BayesianNetworkPlugin</i> class method), 46	<code>sample_hyperparameters()</code> ( <i>WeibullAFTSurvivalAnalysis</i> class method), 441
<code>sample_hyperparameters()</code> ( <i>CoxPHSurvivalAnalysis</i> class method), 442	<code>sample_hyperparameters()</code> ( <i>XGBSurvivalAnalysis</i> class method), 443
<code>sample_hyperparameters()</code> ( <i>CoxTimeSeriesSurvival</i> class method), 444	<code>sample_hyperparameters()</code> ( <i>XGBTimeSeriesSurvival</i> class method), 459
<code>sample_hyperparameters()</code> ( <i>CTGANPlugin</i> class method), 51	<code>sample_hyperparameters_optuna()</code> ( <i>AdsGANPlugin</i> class method), 70
<code>sample_hyperparameters()</code> ( <i>DATETimeToEvent</i> class method), 460	<code>sample_hyperparameters_optuna()</code> ( <i>ARFPlugin</i> class method), 27
<code>sample_hyperparameters()</code> ( <i>DECAFPlugin</i> class method), 90	<code>sample_hyperparameters_optuna()</code> ( <i>BayesianNetworkPlugin</i> class method), 46
<code>sample_hyperparameters()</code> ( <i>DeephitSurvivalAnalysis</i> class method), 442	<code>sample_hyperparameters_optuna()</code> ( <i>CTGANPlugin</i> class method), 51
<code>sample_hyperparameters()</code> ( <i>DPGANPlugin</i> class method), 85	<code>sample_hyperparameters_optuna()</code> ( <i>DECAFPlugin</i> class method), 90
<code>sample_hyperparameters()</code> ( <i>DynamicDeephitTimeSeriesSurvival</i> class method), 458	<code>sample_hyperparameters_optuna()</code> ( <i>DPGANPlugin</i> class method), 85
<code>sample_hyperparameters()</code> ( <i>FourierFlowsPlugin</i> class method), 148	<code>sample_hyperparameters_optuna()</code> ( <i>FourierFlowsPlugin</i> class method), 148
<code>sample_hyperparameters()</code> ( <i>GOGGLEPlugin</i> class method), 24	<code>sample_hyperparameters_optuna()</code> ( <i>GOGGLEPlugin</i> class method), 24
<code>sample_hyperparameters()</code> ( <i>ImageAdsGANPlugin</i> class method), 162	<code>sample_hyperparameters_optuna()</code> ( <i>ImageAdsGANPlugin</i> class method), 162
<code>sample_hyperparameters()</code> ( <i>ImageCGANPlugin</i> class method), 158	<code>sample_hyperparameters_optuna()</code> ( <i>ImageCGANPlugin</i> class method), 158
<code>sample_hyperparameters()</code> ( <i>NormalizingFlowsPlugin</i> class method), 56	<code>sample_hyperparameters_optuna()</code> ( <i>NormalizingFlowsPlugin</i> class method), 56
<code>sample_hyperparameters()</code> ( <i>PATEGANPlugin</i> class method), 74	<code>sample_hyperparameters_optuna()</code> ( <i>PATEGANPlugin</i> class method), 75
<code>sample_hyperparameters()</code> ( <i>Plugin</i> class method), 196	<code>sample_hyperparameters_optuna()</code> ( <i>Plugin</i> class method), 196
<code>sample_hyperparameters()</code> ( <i>PrivBayesPlugin</i> class method), 79	<code>sample_hyperparameters_optuna()</code> ( <i>PrivBayesPlugin</i> class method), 80
<code>sample_hyperparameters()</code> ( <i>RadialGANPlugin</i> class method), 108	<code>sample_hyperparameters_optuna()</code> ( <i>RadialGANPlugin</i> class method), 108
<code>sample_hyperparameters()</code> ( <i>RTVAEPlugin</i> class method), 60	<code>sample_hyperparameters_optuna()</code> ( <i>RTVAEPlugin</i> class method), 60
<code>sample_hyperparameters()</code> ( <i>SurVAEPlugin</i> class method), 134	<code>sample_hyperparameters_optuna()</code> ( <i>SurVAEPlugin</i> class method), 134
<code>sample_hyperparameters()</code> ( <i>SurvivalCTGANPlugin</i> class method), 130	<code>sample_hyperparameters_optuna()</code> ( <i>SurvivalCTGANPlugin</i> class method), 130
<code>sample_hyperparameters()</code> ( <i>SurvivalFunctionTimeToEvent</i> class method), 474	<code>sample_hyperparameters_optuna()</code> ( <i>SurvivalGANPlugin</i> class method), 126
<code>sample_hyperparameters()</code> ( <i>SurvivalGANPlugin</i> class method), 126	<code>sample_hyperparameters_optuna()</code> ( <i>SurvivalNFlowPlugin</i> class method), 138
<code>sample_hyperparameters()</code> ( <i>SurvivalNFlowPlugin</i> class method), 138	<code>sample_hyperparameters_optuna()</code> ( <i>TimeGANPlugin</i> class method), 144
<code>sample_hyperparameters()</code> ( <i>TimeGANPlugin</i> class method), 144	<code>sample_hyperparameters_optuna()</code> ( <i>TimeVAEPlugin</i> class method), 153
<code>sample_hyperparameters()</code> ( <i>TimeVAEPlugin</i> class method), 153	<code>sample_hyperparameters_optuna()</code> ( <i>TVAEPlugin</i> class method), 65
<code>sample_hyperparameters()</code> ( <i>TVAEPlugin</i> class method), 65	<code>sample_marginal()</code> ( <i>CategoricalDistribution</i> method), 201

- sample\_marginal() (*DatetimeDistribution method*), 203
- sample\_marginal() (*Distribution method*), 205
- sample\_marginal() (*FloatDistribution method*), 207
- sample\_marginal() (*IntegerDistribution method*), 210
- sample\_marginal() (*IntLogDistribution method*), 209
- sample\_marginal() (*LogDistribution method*), 212
- sampling\_strategy (*Schema attribute*), 214
- satisfies() (*DataLoader method*), 30
- satisfies() (*GenericDataLoader method*), 32
- satisfies() (*ImageDataLoader method*), 33
- satisfies() (*SurvivalAnalysisDataLoader method*), 35
- satisfies() (*TimeSeriesDataLoader method*), 37
- satisfies() (*TimeSeriesSurvivalDataLoader method*), 39
- save() (*AdsGANPlugin method*), 70
- save() (*ARFPlugin method*), 27
- save() (*BayesianNetworkPlugin method*), 46
- save() (*CoxPHSurvivalAnalysis method*), 442
- save() (*CoxTimeSeriesSurvival method*), 444
- save() (*CTGANPlugin method*), 51
- save() (*DATETimeToEvent method*), 460
- save() (*DECAFPlugin method*), 90
- save() (*DeephitSurvivalAnalysis method*), 442
- save() (*DPGANPlugin method*), 85
- save() (*DynamicDeephitTimeSeriesSurvival method*), 458
- save() (*FourierFlowsPlugin method*), 148
- save() (*GOGGLEPlugin method*), 24
- save() (*ImageAdsGANPlugin method*), 162
- save() (*ImageCGANPlugin method*), 158
- save() (*NormalizingFlowsPlugin method*), 56
- save() (*PATEGAN method*), 71
- save() (*PATEGANPlugin method*), 75
- save() (*Plugin method*), 196
- save() (*PrivBayes method*), 76
- save() (*PrivBayesPlugin method*), 80
- save() (*RadialGANPlugin method*), 108
- save() (*RTVAEPlugin method*), 60
- save() (*Serializable method*), 215
- save() (*SurVAEPlugin method*), 134
- save() (*SurvivalCTGANPlugin method*), 130
- save() (*SurvivalFunctionTimeToEvent method*), 474
- save() (*SurvivalGANPlugin method*), 126
- save() (*SurvivalNFlowPlugin method*), 138
- save() (*Teachers method*), 75
- save() (*TimeGANPlugin method*), 144
- save() (*TimeVAEPlugin method*), 153
- save() (*TVAEPlugin method*), 65
- save() (*WeibullAFTSurvivalAnalysis method*), 441
- save() (*XGBSurvivalAnalysis method*), 443
- save() (*XGBTimeSeriesSurvival method*), 459
- save\_to\_file() (*AdsGANPlugin method*), 70
- save\_to\_file() (*ARFPlugin method*), 27
- save\_to\_file() (*BayesianNetworkPlugin method*), 46
- save\_to\_file() (*CoxPHSurvivalAnalysis method*), 442
- save\_to\_file() (*CoxTimeSeriesSurvival method*), 444
- save\_to\_file() (*CTGANPlugin method*), 51
- save\_to\_file() (*DATETimeToEvent method*), 460
- save\_to\_file() (*DECAFPlugin method*), 90
- save\_to\_file() (*DeephitSurvivalAnalysis method*), 442
- save\_to\_file() (*DPGANPlugin method*), 85
- save\_to\_file() (*DynamicDeephitTimeSeriesSurvival method*), 458
- save\_to\_file() (*FourierFlowsPlugin method*), 148
- save\_to\_file() (*GOGGLEPlugin method*), 24
- save\_to\_file() (*ImageAdsGANPlugin method*), 162
- save\_to\_file() (*ImageCGANPlugin method*), 158
- save\_dict() (*BayesianNetworkPlugin method*), 46
- save\_dict() (*CoxPHSurvivalAnalysis method*), 442
- save\_dict() (*CoxTimeSeriesSurvival method*), 444
- save\_dict() (*CTGANPlugin method*), 51
- save\_dict() (*DATETimeToEvent method*), 460
- save\_dict() (*DECAFPlugin method*), 90
- save\_dict() (*DeephitSurvivalAnalysis method*), 442
- save\_dict() (*DPGANPlugin method*), 85
- save\_dict() (*DynamicDeephitTimeSeriesSurvival method*), 458
- save\_dict() (*FourierFlowsPlugin method*), 148
- save\_dict() (*GOGGLEPlugin method*), 24
- save\_dict() (*ImageAdsGANPlugin method*), 162
- save\_dict() (*ImageCGANPlugin method*), 158
- save\_dict() (*NormalizingFlowsPlugin method*), 56
- save\_dict() (*PATEGAN method*), 71
- save\_dict() (*PATEGANPlugin method*), 75
- save\_dict() (*Plugin method*), 196
- save\_dict() (*PrivBayes method*), 76
- save\_dict() (*PrivBayesPlugin method*), 80
- save\_dict() (*RadialGANPlugin method*), 108
- save\_dict() (*RTVAEPlugin method*), 60
- save\_dict() (*Serializable method*), 215
- save\_dict() (*SurVAEPlugin method*), 134
- save\_dict() (*SurvivalCTGANPlugin method*), 130
- save\_dict() (*SurvivalFunctionTimeToEvent method*), 474
- save\_dict() (*SurvivalGANPlugin method*), 126
- save\_dict() (*SurvivalNFlowPlugin method*), 138
- save\_dict() (*Teachers method*), 75
- save\_dict() (*TimeGANPlugin method*), 144
- save\_dict() (*TimeVAEPlugin method*), 153
- save\_dict() (*TVAEPlugin method*), 65
- save\_dict() (*WeibullAFTSurvivalAnalysis method*), 441
- save\_dict() (*XGBSurvivalAnalysis method*), 443
- save\_dict() (*XGBTimeSeriesSurvival method*), 459



- `save_to_file()` (*NormalizingFlowsPlugin* method), 56  
`save_to_file()` (*PATEGAN* method), 71  
`save_to_file()` (*PATEGANPlugin* method), 75  
`save_to_file()` (*Plugin* method), 196  
`save_to_file()` (*PrivBayes* method), 76  
`save_to_file()` (*PrivBayesPlugin* method), 80  
`save_to_file()` (*RadialGANPlugin* method), 108  
`save_to_file()` (*RTVAEPlugin* method), 60  
`save_to_file()` (*Serializable* method), 215  
`save_to_file()` (*SurVAEPlugin* method), 134  
`save_to_file()` (*SurvivalCTGANPlugin* method), 130  
`save_to_file()` (*SurvivalFunctionTimeToEvent* method), 474  
`save_to_file()` (*SurvivalGANPlugin* method), 126  
`save_to_file()` (*SurvivalNFlowPlugin* method), 138  
`save_to_file()` (*Teachers* method), 75  
`save_to_file()` (*TimeGANPlugin* method), 144  
`save_to_file()` (*TimeVAEPlugin* method), 153  
`save_to_file()` (*TVAEPlugin* method), 65  
`save_to_file()` (*WeibullAFTSurvivalAnalysis* method), 441  
`save_to_file()` (*XGBSurvivalAnalysis* method), 443  
`save_to_file()` (*XGBTimeSeriesSurvival* method), 459  
`Schema` (class in *synthcity.plugins.core.schema*), 213  
`schema()` (*AdsGANPlugin* method), 70  
`schema()` (*ARFPlugin* method), 27  
`schema()` (*BayesianNetworkPlugin* method), 46  
`schema()` (*CategoricalDistribution* class method), 201  
`schema()` (*Constraints* class method), 199  
`schema()` (*CTGANPlugin* method), 51  
`schema()` (*DatetimeDistribution* class method), 203  
`schema()` (*DECAFPlugin* method), 90  
`schema()` (*Distribution* class method), 205  
`schema()` (*DPGANPlugin* method), 85  
`schema()` (*FeatureInfo* class method), 217  
`schema()` (*FloatDistribution* class method), 207  
`schema()` (*FourierFlowsPlugin* method), 148  
`schema()` (*GOGGLEPlugin* method), 24  
`schema()` (*ImageAdsGANPlugin* method), 162  
`schema()` (*ImageCGANPlugin* method), 158  
`schema()` (*IntegerDistribution* class method), 210  
`schema()` (*IntLogDistribution* class method), 209  
`schema()` (*LogDistribution* class method), 212  
`schema()` (*NormalizingFlowsPlugin* method), 56  
`schema()` (*PATEGANPlugin* method), 75  
`schema()` (*Plugin* method), 196  
`schema()` (*PrivBayesPlugin* method), 80  
`schema()` (*RadialGANPlugin* method), 108  
`schema()` (*RTVAEPlugin* method), 61  
`schema()` (*Schema* class method), 214  
`schema()` (*SurVAEPlugin* method), 134  
`schema()` (*SurvivalCTGANPlugin* method), 130  
`schema()` (*SurvivalGANPlugin* method), 126  
`schema()` (*SurvivalNFlowPlugin* method), 138  
`schema()` (*TimeGANPlugin* method), 144  
`schema()` (*TimeVAEPlugin* method), 153  
`schema()` (*TVAEPlugin* method), 65  
`schema_includes()` (*AdsGANPlugin* method), 70  
`schema_includes()` (*ARFPlugin* method), 27  
`schema_includes()` (*BayesianNetworkPlugin* method), 46  
`schema_includes()` (*CTGANPlugin* method), 51  
`schema_includes()` (*DECAFPlugin* method), 90  
`schema_includes()` (*DPGANPlugin* method), 85  
`schema_includes()` (*FourierFlowsPlugin* method), 148  
`schema_includes()` (*GOGGLEPlugin* method), 24  
`schema_includes()` (*ImageAdsGANPlugin* method), 162  
`schema_includes()` (*ImageCGANPlugin* method), 158  
`schema_includes()` (*NormalizingFlowsPlugin* method), 56  
`schema_includes()` (*PATEGANPlugin* method), 75  
`schema_includes()` (*Plugin* method), 196  
`schema_includes()` (*PrivBayesPlugin* method), 80  
`schema_includes()` (*RadialGANPlugin* method), 108  
`schema_includes()` (*RTVAEPlugin* method), 61  
`schema_includes()` (*SurVAEPlugin* method), 134  
`schema_includes()` (*SurvivalCTGANPlugin* method), 130  
`schema_includes()` (*SurvivalGANPlugin* method), 126  
`schema_includes()` (*SurvivalNFlowPlugin* method), 139  
`schema_includes()` (*TimeGANPlugin* method), 144  
`schema_includes()` (*TimeVAEPlugin* method), 153  
`schema_includes()` (*TVAEPlugin* method), 65  
`schema_json()` (*CategoricalDistribution* class method), 201  
`schema_json()` (*Constraints* class method), 199  
`schema_json()` (*DatetimeDistribution* class method), 203  
`schema_json()` (*Distribution* class method), 205  
`schema_json()` (*FeatureInfo* class method), 217  
`schema_json()` (*FloatDistribution* class method), 207  
`schema_json()` (*IntegerDistribution* class method), 211  
`schema_json()` (*IntLogDistribution* class method), 209  
`schema_json()` (*LogDistribution* class method), 212  
`schema_json()` (*Schema* class method), 215  
`score()` (*ConvNet* method), 510  
`score()` (*MLP* method), 242  
`sequential_view()` (*TimeSeriesDataLoader* static method), 37  
`sequential_view()` (*TimeSeriesSurvivalDataLoader* static method), 39  
`Serializable` (class in *synthcity.plugins.core.serializable*), 215  
`set_extra_state()` (*ConditionalDiscriminator* method), 484

- `set_extra_state()` (*ConditionalGenerator method*), 496  
`set_extra_state()` (*ConvNet method*), 510  
`set_extra_state()` (*Decoder method*), 284  
`set_extra_state()` (*DynamicDeepHitLayers method*), 453  
`set_extra_state()` (*Encoder method*), 297  
`set_extra_state()` (*GAN method*), 257  
`set_extra_state()` (*ImageGAN method*), 526  
`set_extra_state()` (*LatentODE method*), 396  
`set_extra_state()` (*LinearLayer method*), 228  
`set_extra_state()` (*MLP method*), 242  
`set_extra_state()` (*NormalizingFlows method*), 272  
`set_extra_state()` (*RadialGAN method*), 101  
`set_extra_state()` (*TabularFlows method*), 368  
`set_extra_state()` (*TabularGAN method*), 340  
`set_extra_state()` (*TabularRadialGAN method*), 119  
`set_extra_state()` (*TabularVAE method*), 354  
`set_extra_state()` (*TimeEventGAN method*), 470  
`set_extra_state()` (*TimeSeriesDecoder method*), 409  
`set_extra_state()` (*TimeSeriesEncoder method*), 422  
`set_extra_state()` (*TimeSeriesGAN method*), 383  
`set_extra_state()` (*TimeSeriesVAE method*), 437  
`set_extra_state()` (*TransformerModel method*), 324  
`set_extra_state()` (*VAE method*), 311  
`shape` (*DataLoader property*), 30  
`shape` (*GenericDataLoader property*), 32  
`shape` (*ImageDataLoader property*), 33  
`shape` (*SurvivalAnalysisDataLoader property*), 35  
`shape` (*TimeSeriesDataLoader property*), 37  
`shape` (*TimeSeriesSurvivalDataLoader property*), 39  
`shape()` (*FlexibleDataset method*), 40  
`share_memory()` (*ConditionalDiscriminator method*), 484  
`share_memory()` (*ConditionalGenerator method*), 497  
`share_memory()` (*ConvNet method*), 510  
`share_memory()` (*Decoder method*), 285  
`share_memory()` (*DynamicDeepHitLayers method*), 454  
`share_memory()` (*Encoder method*), 297  
`share_memory()` (*GAN method*), 258  
`share_memory()` (*ImageGAN method*), 526  
`share_memory()` (*LatentODE method*), 397  
`share_memory()` (*LinearLayer method*), 229  
`share_memory()` (*MLP method*), 242  
`share_memory()` (*NormalizingFlows method*), 272  
`share_memory()` (*RadialGAN method*), 101  
`share_memory()` (*TabularFlows method*), 368  
`share_memory()` (*TabularGAN method*), 340  
`share_memory()` (*TabularRadialGAN method*), 119  
`share_memory()` (*TabularVAE method*), 354  
`share_memory()` (*TimeEventGAN method*), 470  
`share_memory()` (*TimeSeriesDecoder method*), 410  
`share_memory()` (*TimeSeriesEncoder method*), 422  
`share_memory()` (*TimeSeriesGAN method*), 384  
`share_memory()` (*TimeSeriesVAE method*), 437  
`share_memory()` (*TransformerModel method*), 325  
`share_memory()` (*VAE method*), 312  
`standard_performance_output_keys()` (*AugmentationPerformanceEvaluatorLinear static method*), 174  
`standard_performance_output_keys()` (*AugmentationPerformanceEvaluatorMLP static method*), 175  
`standard_performance_output_keys()` (*AugmentationPerformanceEvaluatorXGB static method*), 175  
`standard_performance_output_keys()` (*PerformanceEvaluator static method*), 176  
`standard_performance_output_keys()` (*PerformanceEvaluatorLinear static method*), 177  
`standard_performance_output_keys()` (*PerformanceEvaluatorMLP static method*), 177  
`standard_performance_output_keys()` (*PerformanceEvaluatorXGB static method*), 178  
`state_dict()` (*ConditionalDiscriminator method*), 484  
`state_dict()` (*ConditionalGenerator method*), 497  
`state_dict()` (*ConvNet method*), 510  
`state_dict()` (*Decoder method*), 285  
`state_dict()` (*DynamicDeepHitLayers method*), 454  
`state_dict()` (*Encoder method*), 297  
`state_dict()` (*GAN method*), 258  
`state_dict()` (*ImageGAN method*), 526  
`state_dict()` (*LatentODE method*), 397  
`state_dict()` (*LinearLayer method*), 229  
`state_dict()` (*MLP method*), 243  
`state_dict()` (*NormalizingFlows method*), 272  
`state_dict()` (*RadialGAN method*), 101  
`state_dict()` (*TabularFlows method*), 369  
`state_dict()` (*TabularGAN method*), 340  
`state_dict()` (*TabularRadialGAN method*), 119  
`state_dict()` (*TabularVAE method*), 354  
`state_dict()` (*TimeEventGAN method*), 470  
`state_dict()` (*TimeSeriesDecoder method*), 410  
`state_dict()` (*TimeSeriesEncoder method*), 423  
`state_dict()` (*TimeSeriesGAN method*), 384  
`state_dict()` (*TimeSeriesVAE method*), 438  
`state_dict()` (*TransformerModel method*), 325  
`state_dict()` (*VAE method*), 312  
`StatisticalEvaluator` (class in *synthcity.metrics.eval\_statistical*), 173  
`step` (*DatetimeDistribution attribute*), 203  
`step` (*IntegerDistribution attribute*), 211  
`step` (*IntLogDistribution attribute*), 209  
`suggest_image_classifier_arch()` (in module *synthcity.plugins.core.models.convnet*), 513  
`suggest_image_generator_discriminator_arch()` (in module *synthcity.plugins.core.models.convnet*), 514

SurvVAEPlugin	(class in synthcity.plugins.survival_analysis.plugin_survae), 131	module, 40 synthcity.plugins.core.distribution module, 199
SurvVAEPlugin.Config	(class in synthcity.plugins.survival_analysis.plugin_survae), 132	synthcity.plugins.core.models.convnet module, 474 synthcity.plugins.core.models.flows module, 261
SurvivalAnalysisDataLoader	(class in synthcity.plugins.core.dataloader), 34	synthcity.plugins.core.models.gan module, 246
SurvivalCTGANPlugin	(class in synthcity.plugins.survival_analysis.plugin_survival_ctgan), 127	synthcity.plugins.core.models.image_gan module, 515
SurvivalCTGANPlugin.Config	(class in synthcity.plugins.survival_analysis.plugin_survival_ctgan), 128	synthcity.plugins.core.models.mlp module, 219 synthcity.plugins.core.models.survival_analysis.surv_aft module, 441
SurvivalFunctionTimeToEvent	(class in synthcity.plugins.core.models.time_to_event.tte_survival_analysis), 473	synthcity.plugins.core.models.survival_analysis.surv_coxph module, 441
SurvivalGANPlugin	(class in synthcity.plugins.survival_analysis.plugin_survival_gan), 123	synthcity.plugins.core.models.survival_analysis.surv_deep module, 442 synthcity.plugins.core.models.survival_analysis.surv_xgb module, 443
SurvivalGANPlugin.Config	(class in synthcity.plugins.survival_analysis.plugin_survival_gan), 124	synthcity.plugins.core.models.tabular_encoder module, 215
SurvivalKMDistance	(class in synthcity.metrics.eval_statistical), 173	synthcity.plugins.core.models.tabular_flows module, 358
SurvivalNFlowPlugin	(class in synthcity.plugins.survival_analysis.plugin_survival_nflow), 135	synthcity.plugins.core.models.tabular_gan module, 328 synthcity.plugins.core.models.tabular_vae module, 343
SurvivalNFlowPlugin.Config	(class in synthcity.plugins.survival_analysis.plugin_survival_nflow), 136	synthcity.plugins.core.models.time_series_survival.ts_surv module, 443 synthcity.plugins.core.models.time_series_survival.ts_surv module, 444
synthcity.benchmark	module, 189	synthcity.plugins.core.models.time_series_survival.ts_surv module, 459
synthcity.benchmark.utils	module, 190	synthcity.plugins.core.models.time_to_event.tte_date module, 459
synthcity.metrics.eval_attacks	module, 187	synthcity.plugins.core.models.time_to_event.tte_survival module, 473
synthcity.metrics.eval_detection	module, 178	synthcity.plugins.core.models.transformer module, 315
synthcity.metrics.eval_performance	module, 174	synthcity.plugins.core.models.ts_gan module, 372
synthcity.metrics.eval_privacy	module, 181	synthcity.plugins.core.models.ts_vae module, 387
synthcity.metrics.eval_sanity	module, 165	synthcity.plugins.core.models.vae module, 275
synthcity.metrics.eval_statistical	module, 168	synthcity.plugins.core.plugin module, 193
synthcity.metrics.weighted_metrics	module, 189	synthcity.plugins.core.schema module, 213
synthcity.plugins.core.constraints	module, 197	synthcity.plugins.core.serializable module, 215
synthcity.plugins.core.dataloader	module, 29	synthcity.plugins.domain_adaptation.plugin_radialgan
synthcity.plugins.core.dataset		

- module, 90
  - synthcity.plugins.generic.plugin\_arf
    - module, 24
  - synthcity.plugins.generic.plugin\_bayesian\_network
    - module, 43
  - synthcity.plugins.generic.plugin\_ctgan
    - module, 47
  - synthcity.plugins.generic.plugin\_goggle
    - module, 21
  - synthcity.plugins.generic.plugin\_nflow
    - module, 52
  - synthcity.plugins.generic.plugin\_rtvae
    - module, 57
  - synthcity.plugins.generic.plugin\_tvae
    - module, 61
  - synthcity.plugins.images.plugin\_image\_adsgan
    - module, 158
  - synthcity.plugins.images.plugin\_image\_cgan
    - module, 154
  - synthcity.plugins.privacy.plugin\_adsgan
    - module, 65
  - synthcity.plugins.privacy.plugin\_decaf
    - module, 85
  - synthcity.plugins.privacy.plugin\_dpgan
    - module, 81
  - synthcity.plugins.privacy.plugin\_pategan
    - module, 70
  - synthcity.plugins.privacy.plugin\_privbayes
    - module, 76
  - synthcity.plugins.survival\_analysis.plugin\_survival\_gan
    - module, 131
  - synthcity.plugins.survival\_analysis.plugin\_survival\_radialgan
    - module, 127
  - synthcity.plugins.survival\_analysis.plugin\_survival\_gan
    - module, 123
  - synthcity.plugins.survival\_analysis.plugin\_survival\_nflow
    - module, 135
  - synthcity.plugins.time\_series.plugin\_fflows
    - module, 144
  - synthcity.plugins.time\_series.plugin\_timegan
    - module, 139
  - synthcity.plugins.time\_series.plugin\_timevae
    - module, 149
  - SyntheticDetectionGMM (class in synthcity.metrics.eval\_detection), 179
  - SyntheticDetectionLinear (class in synthcity.metrics.eval\_detection), 179
  - SyntheticDetectionMLP (class in synthcity.metrics.eval\_detection), 180
  - SyntheticDetectionXGB (class in synthcity.metrics.eval\_detection), 181
- T**
- T\_destination (ConditionalDiscriminator attribute), 474
  - T\_destination (ConditionalGenerator attribute), 487
  - T\_destination (ConvNet attribute), 500
  - T\_destination (Decoder attribute), 275
  - T\_destination (DynamicDeepHitLayers attribute), 444
  - T\_destination (Encoder attribute), 288
  - T\_destination (GAN attribute), 248
  - T\_destination (ImageGAN attribute), 517
  - T\_destination (LatentODE attribute), 387
  - T\_destination (LinearLayer attribute), 219
  - T\_destination (MLP attribute), 233
  - T\_destination (NormalizingFlows attribute), 262
  - T\_destination (RadialGAN attribute), 92
  - T\_destination (TabularFlows attribute), 359
  - T\_destination (TabularGAN attribute), 330
  - T\_destination (TabularRadialGAN attribute), 110
  - T\_destination (TabularVAE attribute), 345
  - T\_destination (TimeEventGAN attribute), 460
  - T\_destination (TimeSeriesDecoder attribute), 400
  - T\_destination (TimeSeriesEncoder attribute), 413
  - T\_destination (TimeSeriesGAN attribute), 374
  - T\_destination (TimeSeriesVAE attribute), 428
  - T\_destination (TransformerModel attribute), 315
  - T\_destination (VAE attribute), 302
  - TabularEncoder (class in synthcity.plugins.core.models.tabular\_encoder), 217
  - TabularFlows (class in synthcity.plugins.core.models.tabular\_flows), 358
  - TabularGAN (class in synthcity.plugins.core.models.tabular\_gan), 328
  - TabularRadialGAN (class in synthcity.plugins.domain\_adaptation.plugin\_radialgan), 109
  - TabularVAE (class in synthcity.plugins.core.models.tabular\_vae), 343
  - Teachers (class in synthcity.plugins.privacy.plugin\_pategan), 75
  - TensorDataset (class in synthcity.plugins.core.dataset), 41
  - tensors() (FlexibleDataset method), 40
  - test() (DataLoader method), 30
  - test() (GenericDataLoader method), 32
  - test() (ImageDataLoader method), 33
  - test() (SurvivalAnalysisDataLoader method), 35
  - test() (TimeSeriesDataLoader method), 37
  - test() (TimeSeriesSurvivalDataLoader method), 39
  - TimeEventGAN (class in synthcity.plugins.core.models.time\_to\_event.tte\_date), 460
  - TimeGANPlugin (class in synthcity.plugins.time\_series.plugin\_timegan), 139
  - TimeGANPlugin.Config (class in synthcity.plugins.time\_series.plugin\_timegan), 139



[ity.plugins.time\\_series.plugin\\_timegan](#)),  
[141](#)  
[TimeSeriesBinEncoder](#) (class in [synthcity.plugins.core.models.tabular\\_encoder](#)),  
[218](#)  
[TimeSeriesDataLoader](#) (class in [synthcity.plugins.core.dataloader](#)), [35](#)  
[TimeSeriesDecoder](#) (class in [synthcity.plugins.core.models.ts\\_vae](#)), [400](#)  
[TimeSeriesEncoder](#) (class in [synthcity.plugins.core.models.ts\\_vae](#)), [413](#)  
[TimeSeriesGAN](#) (class in [synthcity.plugins.core.models.ts\\_gan](#)), [372](#)  
[TimeSeriesSurvivalDataLoader](#) (class in [synthcity.plugins.core.dataloader](#)), [37](#)  
[TimeSeriesTabularEncoder](#) (class in [synthcity.plugins.core.models.tabular\\_encoder](#)),  
[218](#)  
[TimeSeriesVAE](#) (class in [synthcity.plugins.core.models.ts\\_vae](#)), [426](#)  
[TimeVAEPlugin](#) (class in [synthcity.plugins.time\\_series.plugin\\_timevae](#)), [149](#)  
[TimeVAEPlugin.Config](#) (class in [synthcity.plugins.time\\_series.plugin\\_timevae](#)), [151](#)  
[to\(\)](#) ([ConditionalDiscriminator](#) method), [484](#)  
[to\(\)](#) ([ConditionalGenerator](#) method), [497](#)  
[to\(\)](#) ([ConvNet](#) method), [511](#)  
[to\(\)](#) ([Decoder](#) method), [285](#)  
[to\(\)](#) ([DynamicDeepHitLayers](#) method), [454](#)  
[to\(\)](#) ([Encoder](#) method), [298](#)  
[to\(\)](#) ([GAN](#) method), [258](#)  
[to\(\)](#) ([ImageGAN](#) method), [527](#)  
[to\(\)](#) ([LatentODE](#) method), [397](#)  
[to\(\)](#) ([LinearLayer](#) method), [229](#)  
[to\(\)](#) ([MLP](#) method), [243](#)  
[to\(\)](#) ([NormalizingFlows](#) method), [272](#)  
[to\(\)](#) ([RadialGAN](#) method), [102](#)  
[to\(\)](#) ([TabularFlows](#) method), [369](#)  
[to\(\)](#) ([TabularGAN](#) method), [341](#)  
[to\(\)](#) ([TabularRadialGAN](#) method), [120](#)  
[to\(\)](#) ([TabularVAE](#) method), [355](#)  
[to\(\)](#) ([TimeEventGAN](#) method), [471](#)  
[to\(\)](#) ([TimeSeriesDecoder](#) method), [410](#)  
[to\(\)](#) ([TimeSeriesEncoder](#) method), [423](#)  
[to\(\)](#) ([TimeSeriesGAN](#) method), [384](#)  
[to\(\)](#) ([TimeSeriesVAE](#) method), [438](#)  
[to\(\)](#) ([TransformerModel](#) method), [325](#)  
[to\(\)](#) ([VAE](#) method), [312](#)  
[to\\_empty\(\)](#) ([ConditionalDiscriminator](#) method), [486](#)  
[to\\_empty\(\)](#) ([ConditionalGenerator](#) method), [499](#)  
[to\\_empty\(\)](#) ([ConvNet](#) method), [512](#)  
[to\\_empty\(\)](#) ([Decoder](#) method), [287](#)  
[to\\_empty\(\)](#) ([DynamicDeepHitLayers](#) method), [456](#)  
[to\\_empty\(\)](#) ([Encoder](#) method), [300](#)  
[to\\_empty\(\)](#) ([GAN](#) method), [260](#)  
[to\\_empty\(\)](#) ([ImageGAN](#) method), [528](#)  
[to\\_empty\(\)](#) ([LatentODE](#) method), [399](#)  
[to\\_empty\(\)](#) ([LinearLayer](#) method), [231](#)  
[to\\_empty\(\)](#) ([MLP](#) method), [245](#)  
[to\\_empty\(\)](#) ([NormalizingFlows](#) method), [274](#)  
[to\\_empty\(\)](#) ([RadialGAN](#) method), [103](#)  
[to\\_empty\(\)](#) ([TabularFlows](#) method), [371](#)  
[to\\_empty\(\)](#) ([TabularGAN](#) method), [342](#)  
[to\\_empty\(\)](#) ([TabularRadialGAN](#) method), [121](#)  
[to\\_empty\(\)](#) ([TabularVAE](#) method), [356](#)  
[to\\_empty\(\)](#) ([TimeEventGAN](#) method), [472](#)  
[to\\_empty\(\)](#) ([TimeSeriesDecoder](#) method), [412](#)  
[to\\_empty\(\)](#) ([TimeSeriesEncoder](#) method), [425](#)  
[to\\_empty\(\)](#) ([TimeSeriesGAN](#) method), [386](#)  
[to\\_empty\(\)](#) ([TimeSeriesVAE](#) method), [440](#)  
[to\\_empty\(\)](#) ([TransformerModel](#) method), [327](#)  
[to\\_empty\(\)](#) ([VAE](#) method), [314](#)  
[total\\_loss\(\)](#) ([DynamicDeepHitModel](#) method), [458](#)  
[train\(\)](#) ([ConditionalDiscriminator](#) method), [486](#)  
[train\(\)](#) ([ConditionalGenerator](#) method), [499](#)  
[train\(\)](#) ([ConvNet](#) method), [512](#)  
[train\(\)](#) ([DataLoader](#) method), [30](#)  
[train\(\)](#) ([Decoder](#) method), [287](#)  
[train\(\)](#) ([DynamicDeepHitLayers](#) method), [456](#)  
[train\(\)](#) ([Encoder](#) method), [300](#)  
[train\(\)](#) ([GAN](#) method), [260](#)  
[train\(\)](#) ([GenericDataLoader](#) method), [32](#)  
[train\(\)](#) ([ImageDataLoader](#) method), [33](#)  
[train\(\)](#) ([ImageGAN](#) method), [528](#)  
[train\(\)](#) ([LatentODE](#) method), [399](#)  
[train\(\)](#) ([LinearLayer](#) method), [231](#)  
[train\(\)](#) ([MLP](#) method), [245](#)  
[train\(\)](#) ([NormalizingFlows](#) method), [274](#)  
[train\(\)](#) ([RadialGAN](#) method), [103](#)  
[train\(\)](#) ([SurvivalAnalysisDataLoader](#) method), [35](#)  
[train\(\)](#) ([TabularFlows](#) method), [371](#)  
[train\(\)](#) ([TabularGAN](#) method), [342](#)  
[train\(\)](#) ([TabularRadialGAN](#) method), [122](#)  
[train\(\)](#) ([TabularVAE](#) method), [356](#)  
[train\(\)](#) ([TimeEventGAN](#) method), [472](#)  
[train\(\)](#) ([TimeSeriesDataLoader](#) method), [37](#)  
[train\(\)](#) ([TimeSeriesDecoder](#) method), [412](#)  
[train\(\)](#) ([TimeSeriesEncoder](#) method), [425](#)  
[train\(\)](#) ([TimeSeriesGAN](#) method), [386](#)  
[train\(\)](#) ([TimeSeriesSurvivalDataLoader](#) method), [39](#)  
[train\(\)](#) ([TimeSeriesVAE](#) method), [440](#)  
[train\(\)](#) ([TransformerModel](#) method), [327](#)  
[train\(\)](#) ([VAE](#) method), [314](#)  
[training](#) ([ConditionalDiscriminator](#) attribute), [486](#)  
[training](#) ([ConditionalGenerator](#) attribute), [499](#)  
[training](#) ([ConvNet](#) attribute), [513](#)  
[training](#) ([Decoder](#) attribute), [287](#)  
[training](#) ([DynamicDeepHitLayers](#) attribute), [456](#)



- training (*Encoder attribute*), 300
- training (*GAN attribute*), 260
- training (*ImageGAN attribute*), 529
- training (*LatentODE attribute*), 399
- training (*LinearLayer attribute*), 231
- training (*MLP attribute*), 245
- training (*NormalizingFlows attribute*), 274
- training (*RadialGAN attribute*), 104
- training (*TabularFlows attribute*), 371
- training (*TabularGAN attribute*), 343
- training (*TabularRadialGAN attribute*), 122
- training (*TabularVAE attribute*), 357
- training (*TimeEventGAN attribute*), 473
- training (*TimeSeriesDecoder attribute*), 412
- training (*TimeSeriesEncoder attribute*), 425
- training (*TimeSeriesGAN attribute*), 386
- training (*TimeSeriesVAE attribute*), 440
- training (*TransformerModel attribute*), 327
- training (*VAE attribute*), 314
- training\_schema() (*AdsGANPlugin method*), 70
- training\_schema() (*ARFPlugin method*), 28
- training\_schema() (*BayesianNetworkPlugin method*), 47
- training\_schema() (*CTGANPlugin method*), 51
- training\_schema() (*DECAFPlugin method*), 90
- training\_schema() (*DPGANPlugin method*), 85
- training\_schema() (*FourierFlowsPlugin method*), 148
- training\_schema() (*GOGGLEPlugin method*), 24
- training\_schema() (*ImageAdsGANPlugin method*), 163
- training\_schema() (*ImageCGANPlugin method*), 158
- training\_schema() (*NormalizingFlowsPlugin method*), 56
- training\_schema() (*PATEGANPlugin method*), 75
- training\_schema() (*Plugin method*), 196
- training\_schema() (*PrivBayesPlugin method*), 80
- training\_schema() (*RadialGANPlugin method*), 108
- training\_schema() (*RTVAEPlugin method*), 61
- training\_schema() (*SurVAEPlugin method*), 135
- training\_schema() (*SurvivalCTGANPlugin method*), 130
- training\_schema() (*SurvivalGANPlugin method*), 126
- training\_schema() (*SurvivalNFlowPlugin method*), 139
- training\_schema() (*TimeGANPlugin method*), 144
- training\_schema() (*TimeVAEPlugin method*), 153
- training\_schema() (*TVAEPlugin method*), 65
- trans\_feature\_types (*FeatureInfo attribute*), 217
- transform (*FeatureInfo attribute*), 217
- transform() (*BinEncoder method*), 216
- transform() (*TabularEncoder method*), 218
- transform() (*TimeSeriesBinEncoder method*), 218
- transform() (*TimeSeriesTabularEncoder method*), 219
- transform\_observation\_times() (*TimeSeriesTabularEncoder method*), 219
- transform\_static() (*TimeSeriesTabularEncoder method*), 219
- transform\_temporal() (*TimeSeriesTabularEncoder method*), 219
- transformed\_features (*FeatureInfo attribute*), 217
- TransformerModel (class in *synthcity.plugins.core.models.transformer*), 315
- TVAEPlugin (class in *synthcity.plugins.generic.plugin\_tvae*), 61
- TVAEPlugin.Config (class in *synthcity.plugins.generic.plugin\_tvae*), 62
- type() (*AdsGANPlugin static method*), 70
- type() (*AlphaPrecision static method*), 169
- type() (*ARFPlugin static method*), 28
- type() (*AttackEvaluator static method*), 187
- type() (*AugmentationPerformanceEvaluatorLinear static method*), 174
- type() (*AugmentationPerformanceEvaluatorMLP static method*), 175
- type() (*AugmentationPerformanceEvaluatorXGB static method*), 175
- type() (*BasicMetricEvaluator static method*), 165
- type() (*BayesianNetworkPlugin static method*), 47
- type() (*ChiSquaredTest static method*), 169
- type() (*CloseValuesProbability static method*), 166
- type() (*CommonRowsProportion static method*), 166
- type() (*ConditionalDiscriminator method*), 486
- type() (*ConditionalGenerator method*), 499
- type() (*ConvNet method*), 513
- type() (*CTGANPlugin static method*), 51
- type() (*DataLeakageLinear static method*), 188
- type() (*DataLeakageMLP static method*), 188
- type() (*DataLeakageXGB static method*), 189
- type() (*DataLoader method*), 30
- type() (*DataMismatchScore static method*), 167
- type() (*DECAFPlugin static method*), 90
- type() (*Decoder method*), 287
- type() (*DeltaPresence static method*), 182
- type() (*DetectionEvaluator static method*), 179
- type() (*DistantValuesProbability static method*), 167
- type() (*DomiasMIA static method*), 182
- type() (*DomiasMIABNAF static method*), 183
- type() (*DomiasMIAKDE static method*), 183
- type() (*DomiasMIAPrior static method*), 184
- type() (*DPGANPlugin static method*), 85
- type() (*DynamicDeepHitLayers method*), 456
- type() (*Encoder method*), 300
- type() (*FeatureImportanceRankDistance static method*), 176
- type() (*FourierFlowsPlugin static method*), 148
- type() (*FrechetInceptionDistance static method*), 170
- type() (*GAN method*), 260

type() (*GenericDataLoader* method), 32  
 type() (*GOGGLEPlugin* static method), 24  
 type() (*IdentifiabilityScore* static method), 184  
 type() (*ImageAdsGANPlugin* static method), 163  
 type() (*ImageCGANPlugin* static method), 158  
 type() (*ImageDataLoader* method), 34  
 type() (*ImageGAN* method), 529  
 type() (*InverseKLDivergence* static method), 170  
 type() (*JensenShannonDistance* static method), 171  
 type() (*kAnonymization* static method), 185  
 type() (*kMap* static method), 186  
 type() (*KolmogorovSmirnovTest* static method), 171  
 type() (*LatentODE* method), 399  
 type() (*IDiversityDistinct* static method), 186  
 type() (*LinearLayer* method), 231  
 type() (*MaximumMeanDiscrepancy* static method), 172  
 type() (*MLP* method), 245  
 type() (*NearestSyntheticNeighborDistance* static method), 168  
 type() (*NormalizingFlows* method), 274  
 type() (*NormalizingFlowsPlugin* static method), 56  
 type() (*PATEGANPlugin* static method), 75  
 type() (*PerformanceEvaluator* static method), 176  
 type() (*PerformanceEvaluatorLinear* static method), 177  
 type() (*PerformanceEvaluatorMLP* static method), 178  
 type() (*PerformanceEvaluatorXGB* static method), 178  
 type() (*Plugin* static method), 196  
 type() (*PRDCScore* static method), 172  
 type() (*PrivacyEvaluator* static method), 185  
 type() (*PrivBayesPlugin* static method), 80  
 type() (*RadialGAN* method), 104  
 type() (*RadialGANPlugin* static method), 109  
 type() (*RTVAEPlugin* static method), 61  
 type() (*StatisticalEvaluator* static method), 173  
 type() (*SurVAEPlugin* static method), 135  
 type() (*SurvivalAnalysisDataLoader* method), 35  
 type() (*SurvivalCTGANPlugin* static method), 131  
 type() (*SurvivalGANPlugin* static method), 127  
 type() (*SurvivalKMDistance* static method), 173  
 type() (*SurvivalNFlowPlugin* static method), 139  
 type() (*SyntheticDetectionGMM* static method), 179  
 type() (*SyntheticDetectionLinear* static method), 180  
 type() (*SyntheticDetectionMLP* static method), 181  
 type() (*SyntheticDetectionXGB* static method), 181  
 type() (*TabularFlows* method), 371  
 type() (*TabularGAN* method), 343  
 type() (*TabularRadialGAN* method), 122  
 type() (*TabularVAE* method), 357  
 type() (*TimeEventGAN* method), 473  
 type() (*TimeGANPlugin* static method), 144  
 type() (*TimeSeriesDataLoader* method), 37  
 type() (*TimeSeriesDecoder* method), 412  
 type() (*TimeSeriesEncoder* method), 425

type() (*TimeSeriesGAN* method), 386  
 type() (*TimeSeriesSurvivalDataLoader* method), 40  
 type() (*TimeSeriesVAE* method), 440  
 type() (*TimeVAEPlugin* static method), 153  
 type() (*TransformerModel* method), 327  
 type() (*TVAEPlugin* static method), 65  
 type() (*VAE* method), 314  
 type() (*WassersteinDistance* static method), 174  
 types() (*PluginLoader* method), 197

## U

unique\_temporal\_features() (*TimeSeriesDataLoader* static method), 37  
 unique\_temporal\_features() (*TimeSeriesSurvivalDataLoader* static method), 40  
 unmask\_temporal\_data() (*TimeSeriesDataLoader* static method), 37  
 unmask\_temporal\_data() (*TimeSeriesSurvivalDataLoader* static method), 40  
 unpack() (*DataLoader* method), 30  
 unpack() (*GenericDataLoader* method), 32  
 unpack() (*ImageDataLoader* method), 34  
 unpack() (*SurvivalAnalysisDataLoader* method), 35  
 unpack() (*TimeSeriesDataLoader* method), 37  
 unpack() (*TimeSeriesSurvivalDataLoader* method), 40  
 unpack\_and\_decorate() (*TimeSeriesDataLoader* method), 37  
 unpack\_and\_decorate() (*TimeSeriesSurvivalDataLoader* method), 40  
 unpack\_raw\_data() (*TimeSeriesDataLoader* static method), 37  
 unpack\_raw\_data() (*TimeSeriesSurvivalDataLoader* static method), 40  
 update\_forward\_refs() (*CategoricalDistribution* class method), 201  
 update\_forward\_refs() (*Constraints* class method), 199  
 update\_forward\_refs() (*DatetimeDistribution* class method), 203  
 update\_forward\_refs() (*Distribution* class method), 205  
 update\_forward\_refs() (*FeatureInfo* class method), 217  
 update\_forward\_refs() (*FloatDistribution* class method), 207  
 update\_forward\_refs() (*IntegerDistribution* class method), 211  
 update\_forward\_refs() (*IntLogDistribution* class method), 209  
 update\_forward\_refs() (*LogDistribution* class method), 212  
 update\_forward\_refs() (*Schema* class method), 215  
 use\_cache() (*AlphaPrecision* method), 169  
 use\_cache() (*AttackEvaluator* method), 187

- [use\\_cache\(\)](#) (*AugmentationPerformanceEvaluatorLinear* method), 174  
[use\\_cache\(\)](#) (*AugmentationPerformanceEvaluatorMLP* method), 175  
[use\\_cache\(\)](#) (*AugmentationPerformanceEvaluatorXGB* method), 175  
[use\\_cache\(\)](#) (*BasicMetricEvaluator* method), 165  
[use\\_cache\(\)](#) (*ChiSquaredTest* method), 169  
[use\\_cache\(\)](#) (*CloseValuesProbability* method), 166  
[use\\_cache\(\)](#) (*CommonRowsProportion* method), 166  
[use\\_cache\(\)](#) (*DataLeakageLinear* method), 188  
[use\\_cache\(\)](#) (*DataLeakageMLP* method), 188  
[use\\_cache\(\)](#) (*DataLeakageXGB* method), 189  
[use\\_cache\(\)](#) (*DataMismatchScore* method), 167  
[use\\_cache\(\)](#) (*DeltaPresence* method), 182  
[use\\_cache\(\)](#) (*DetectionEvaluator* method), 179  
[use\\_cache\(\)](#) (*DistantValuesProbability* method), 167  
[use\\_cache\(\)](#) (*DomiasMIA* method), 182  
[use\\_cache\(\)](#) (*DomiasMIABNAF* method), 183  
[use\\_cache\(\)](#) (*DomiasMIAKDE* method), 183  
[use\\_cache\(\)](#) (*DomiasMIAPrior* method), 184  
[use\\_cache\(\)](#) (*FeatureImportanceRankDistance* method), 176  
[use\\_cache\(\)](#) (*FrechetInceptionDistance* method), 170  
[use\\_cache\(\)](#) (*IdentifiabilityScore* method), 184  
[use\\_cache\(\)](#) (*InverseKLDivergence* method), 171  
[use\\_cache\(\)](#) (*JensenShannonDistance* method), 171  
[use\\_cache\(\)](#) (*kAnonymization* method), 185  
[use\\_cache\(\)](#) (*kMap* method), 186  
[use\\_cache\(\)](#) (*KolmogorovSmirnovTest* method), 171  
[use\\_cache\(\)](#) (*IDiversityDistinct* method), 186  
[use\\_cache\(\)](#) (*MaximumMeanDiscrepancy* method), 172  
[use\\_cache\(\)](#) (*NearestSyntheticNeighborDistance* method), 168  
[use\\_cache\(\)](#) (*PerformanceEvaluator* method), 176  
[use\\_cache\(\)](#) (*PerformanceEvaluatorLinear* method), 177  
[use\\_cache\(\)](#) (*PerformanceEvaluatorMLP* method), 178  
[use\\_cache\(\)](#) (*PerformanceEvaluatorXGB* method), 178  
[use\\_cache\(\)](#) (*PRDCScore* method), 172  
[use\\_cache\(\)](#) (*PrivacyEvaluator* method), 185  
[use\\_cache\(\)](#) (*StatisticalEvaluator* method), 173  
[use\\_cache\(\)](#) (*SurvivalKMDistance* method), 173  
[use\\_cache\(\)](#) (*SyntheticDetectionGMM* method), 179  
[use\\_cache\(\)](#) (*SyntheticDetectionLinear* method), 180  
[use\\_cache\(\)](#) (*SyntheticDetectionMLP* method), 181  
[use\\_cache\(\)](#) (*SyntheticDetectionXGB* method), 181  
[use\\_cache\(\)](#) (*WassersteinDistance* method), 174  
[usefulness\\_minus\\_target\(\)](#) (in module *synthcity.plugins.privacy.plugin\_privbayses*), 80
- V**  
[VAE](#) (class in *synthcity.plugins.core.models.vae*), 301  
[validate\(\)](#) (*CategoricalDistribution* class method), 201  
[validate\(\)](#) (*Constraints* class method), 199  
[validate\(\)](#) (*DatetimeDistribution* class method), 203  
[validate\(\)](#) (*Distribution* class method), 205  
[validate\(\)](#) (*FeatureInfo* class method), 217  
[validate\(\)](#) (*FloatDistribution* class method), 207  
[validate\(\)](#) (*IntegerDistribution* class method), 211  
[validate\(\)](#) (*IntLogDistribution* class method), 209  
[validate\(\)](#) (*LogDistribution* class method), 212  
[validate\(\)](#) (*Schema* class method), 215  
[validate\\_assignment](#) (*AdsGANPlugin.Config* attribute), 67  
[validate\\_assignment](#) (*ARFPlugin.Config* attribute), 25  
[validate\\_assignment](#) (*BayesianNetworkPlugin.Config* attribute), 44  
[validate\\_assignment](#) (*CTGANPlugin.Config* attribute), 49  
[validate\\_assignment](#) (*DECAFPPlugin.Config* attribute), 87  
[validate\\_assignment](#) (*DPGANPlugin.Config* attribute), 83  
[validate\\_assignment](#) (*FourierFlowsPlugin.Config* attribute), 146  
[validate\\_assignment](#) (*GOGGLEPlugin.Config* attribute), 22  
[validate\\_assignment](#) (*ImageAdsGANPlugin.Config* attribute), 160  
[validate\\_assignment](#) (*ImageCGANPlugin.Config* attribute), 155  
[validate\\_assignment](#) (*NormalizingFlowsPlugin.Config* attribute), 54  
[validate\\_assignment](#) (*PATEGANPlugin.Config* attribute), 72  
[validate\\_assignment](#) (*Plugin.Config* attribute), 194  
[validate\\_assignment](#) (*PrivBayesPlugin.Config* attribute), 77  
[validate\\_assignment](#) (*RadialGANPlugin.Config* attribute), 106  
[validate\\_assignment](#) (*RTVAEPlugin.Config* attribute), 58  
[validate\\_assignment](#) (*SurVAEPlugin.Config* attribute), 132  
[validate\\_assignment](#) (*SurvivalCTGANPlugin.Config* attribute), 128  
[validate\\_assignment](#) (*SurvivalGANPlugin.Config* attribute), 124  
[validate\\_assignment](#) (*SurvivalNFlowPlugin.Config* attribute), 136  
[validate\\_assignment](#) (*TimeGANPlugin.Config* attribute), 141  
[validate\\_assignment](#) (*TimeVAEPlugin.Config* attribute), 151  
[validate\\_assignment](#) (*TVAEPlugin.Config* attribute), 62



[values \(DataLoader property\)](#), 30  
[values \(GenericDataLoader property\)](#), 32  
[values \(ImageDataLoader property\)](#), 34  
[values \(SurvivalAnalysisDataLoader property\)](#), 35  
[values \(TimeSeriesDataLoader property\)](#), 37  
[values \(TimeSeriesSurvivalDataLoader property\)](#), 40  
[version\(\) \(AdsGANPlugin static method\)](#), 70  
[version\(\) \(ARFPlugin static method\)](#), 28  
[version\(\) \(BayesianNetworkPlugin static method\)](#), 47  
[version\(\) \(CoxPHSurvivalAnalysis static method\)](#), 442  
[version\(\) \(CoxTimeSeriesSurvival static method\)](#), 444  
[version\(\) \(CTGANPlugin static method\)](#), 52  
[version\(\) \(DATETimeToEvent static method\)](#), 460  
[version\(\) \(DECAFPlugin static method\)](#), 90  
[version\(\) \(DeephitSurvivalAnalysis static method\)](#), 442  
[version\(\) \(DPGANPlugin static method\)](#), 85  
[version\(\) \(DynamicDeephitTimeSeriesSurvival static method\)](#), 458  
[version\(\) \(FourierFlowsPlugin static method\)](#), 148  
[version\(\) \(GOGGLEPlugin static method\)](#), 24  
[version\(\) \(ImageAdsGANPlugin static method\)](#), 163  
[version\(\) \(ImageCGANPlugin static method\)](#), 158  
[version\(\) \(NormalizingFlowsPlugin static method\)](#), 57  
[version\(\) \(PATEGAN static method\)](#), 71  
[version\(\) \(PATEGANPlugin static method\)](#), 75  
[version\(\) \(Plugin static method\)](#), 196  
[version\(\) \(PrivBayes static method\)](#), 76  
[version\(\) \(PrivBayesPlugin static method\)](#), 80  
[version\(\) \(RadialGANPlugin static method\)](#), 109  
[version\(\) \(RTVAEPlugin static method\)](#), 61  
[version\(\) \(Serializable static method\)](#), 215  
[version\(\) \(SurVAEPlugin static method\)](#), 135  
[version\(\) \(SurvivalCTGANPlugin static method\)](#), 131  
[version\(\) \(SurvivalFunctionTimeToEvent static method\)](#), 474  
[version\(\) \(SurvivalGANPlugin static method\)](#), 127  
[version\(\) \(SurvivalNFlowPlugin static method\)](#), 139  
[version\(\) \(Teachers static method\)](#), 75  
[version\(\) \(TimeGANPlugin static method\)](#), 144  
[version\(\) \(TimeVAEPlugin static method\)](#), 153  
[version\(\) \(TVAEPlugin static method\)](#), 65  
[version\(\) \(WeibullAFTSurvivalAnalysis static method\)](#), 441  
[version\(\) \(XGBSurvivalAnalysis static method\)](#), 443  
[version\(\) \(XGBTimeSeriesSurvival static method\)](#), 459

## W

[WassersteinDistance \(class in synthcity.metrics.eval\\_statistical\)](#), 173  
[WeibullAFTSurvivalAnalysis \(class in synthcity.plugins.core.models.survival\\_analysis.surv\\_aft\)](#), 441  
[WeightedMetrics \(class in synthcity.metrics.weighted\\_metrics\)](#), 189

[weights\\_init\(\) \(in module synthcity.plugins.core.models.image\\_gan\)](#), 529

## X

[XGBSurvivalAnalysis \(class in synthcity.plugins.core.models.survival\\_analysis.surv\\_xgb\)](#), 443  
[XGBTimeSeriesSurvival \(class in synthcity.plugins.core.models.time\\_series\\_survival.ts\\_surv\\_xgb\)](#), 459  
[xpu\(\) \(ConditionalDiscriminator method\)](#), 486  
[xpu\(\) \(ConditionalGenerator method\)](#), 499  
[xpu\(\) \(ConvNet method\)](#), 513  
[xpu\(\) \(Decoder method\)](#), 287  
[xpu\(\) \(DynamicDeepHitLayers method\)](#), 456  
[xpu\(\) \(Encoder method\)](#), 300  
[xpu\(\) \(GAN method\)](#), 260  
[xpu\(\) \(ImageGAN method\)](#), 529  
[xpu\(\) \(LatentODE method\)](#), 399  
[xpu\(\) \(LinearLayer method\)](#), 231  
[xpu\(\) \(MLP method\)](#), 245  
[xpu\(\) \(NormalizingFlows method\)](#), 274  
[xpu\(\) \(RadialGAN method\)](#), 104  
[xpu\(\) \(TabularFlows method\)](#), 371  
[xpu\(\) \(TabularGAN method\)](#), 343  
[xpu\(\) \(TabularRadialGAN method\)](#), 122  
[xpu\(\) \(TabularVAE method\)](#), 357  
[xpu\(\) \(TimeEventGAN method\)](#), 473  
[xpu\(\) \(TimeSeriesDecoder method\)](#), 412  
[xpu\(\) \(TimeSeriesEncoder method\)](#), 425  
[xpu\(\) \(TimeSeriesGAN method\)](#), 386  
[xpu\(\) \(TimeSeriesVAE method\)](#), 440  
[xpu\(\) \(TransformerModel method\)](#), 327  
[xpu\(\) \(VAE method\)](#), 314

## Z

[zero\\_grad\(\) \(ConditionalDiscriminator method\)](#), 487  
[zero\\_grad\(\) \(ConditionalGenerator method\)](#), 500  
[zero\\_grad\(\) \(ConvNet method\)](#), 513  
[zero\\_grad\(\) \(Decoder method\)](#), 288  
[zero\\_grad\(\) \(DynamicDeepHitLayers method\)](#), 457  
[zero\\_grad\(\) \(Encoder method\)](#), 301  
[zero\\_grad\(\) \(GAN method\)](#), 261  
[zero\\_grad\(\) \(ImageGAN method\)](#), 529  
[zero\\_grad\(\) \(LatentODE method\)](#), 400  
[zero\\_grad\(\) \(LinearLayer method\)](#), 232  
[zero\\_grad\(\) \(MLP method\)](#), 246  
[zero\\_grad\(\) \(NormalizingFlows method\)](#), 275  
[zero\\_grad\(\) \(RadialGAN method\)](#), 104  
[zero\\_grad\(\) \(TabularFlows method\)](#), 372  
[zero\\_grad\(\) \(TabularGAN method\)](#), 343  
[zero\\_grad\(\) \(TabularRadialGAN method\)](#), 122  
[zero\\_grad\(\) \(TabularVAE method\)](#), 357  
[zero\\_grad\(\) \(TimeEventGAN method\)](#), 473

`zero_grad()` (*TimeSeriesDecoder method*), 413  
`zero_grad()` (*TimeSeriesEncoder method*), 426  
`zero_grad()` (*TimeSeriesGAN method*), 387  
`zero_grad()` (*TimeSeriesVAE method*), 441  
`zero_grad()` (*TransformerModel method*), 328  
`zero_grad()` (*VAE method*), 315